THE ORIC-1 and how to get the most from it

IAN SINCLAIR

The ORIC-I

and how to get the most from it

The ORIC-I and how to get the most from it

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Publishing Limited – Technical Books Division
Frogmore, St Albans, Herts AL2 2NF
and
36 Golden Square, London W1R 4AH
515 Madison Avenue, New York, NY 10022, USA
117 York Street, Sydney, NSW 2000, Australia
60 International Boulevard, Rexdale, Ontario R9W 6J2, Canada
61 Beach Road, Auckland, New Zealand

British Library Cataloguing in Publication Data
Sinclair, Ian, R.
The ORIC-1 and how to get the most from it.
1. ORIC 1 (Computer)
I. Title
001.64′04 QA76.8.0/

ISBN 0–246–12130–0

First published in Great Britain 1983 by Granada Publishing
Reprinted 1983

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed in Great Britain by Mackays of Chatham, Kent

Contents

Preface

The astonishing boom in home computing which started in 1982 has been considerably accelerated by the introduction of the ORIC-1. This machine must rank among the most remarkable value-for-money small computers that have been launched, and this book is dedicated to the buyer who has absolutely no previous knowledge of computing.

Unlike so many electrical devices, a computer cannot simply be plugged in and switched on – it is *not* like the games machines which are sometimes described as 'computers'. One of the considerable pleasures of computer ownership is the sense of achievement that you get by having programmed the machine to do what you want. It's true to say that the more you understand your computer, the more use and pleasure you can get out of it. This book is therefore designed to allow you to get as much as possible from your ORIC-1 while you are still learning the business of computing.

The manual for a computer must, of course, contain brief details of all the instructions that the computer will obey. The beginner, however, always needs much more detailed instructions than a manual can ever find space for, and this book is intended to fill the gap. By the time you have finished this book, you will be familiar with the ways in which the ORIC can be programmed. You will, furthermore, be able to make much more imaginative use of the machine, and of all the add-ons and books of more advanced programming that will follow it.

While the text of this book was being written, programs were tested and printed using a very early model of the ORIC-1. All of the programs which appear in the book have been entered into my machine and successfully run.

A book like this is always the result of a great deal of effort by a large number of people. I must start, as always, with Richard Miles, of Granada Publishing Ltd, for continual encouragement and

tireless negotiation with manufacturers. I particularly want to acknowledge the outstanding co-operation of ORIC International and Tangerine Ltd, particularly the contributions of Paul Kaufman and Peter Halford. In addition, the early delivery of my ORIC was undoubtedly due to the efforts of Judith Patten and Pam Durkie of Judith Patten Public Relations. As always, the book could not have been produced on such a tight schedule without the help, encouragement, and assistance of my wife.

Ian Sinclair

Chapter One
Connections

Congratulations on your choice! Now that you possess one of the most remarkable small computers ever designed, you will be anxious to get the best out of it. Don't rush it, though. Before you can use the ORIC-1 really effectively, you have to set it up correctly, and that involves more than just plugging it in. You also need to be able to record and replay the ORIC signals, using a cassette recorder. A recorder, therefore, is an essential extra, along with a TV receiver so that you can see the results of your work with the ORIC. Don't try to use a stereo 'hi-fi' type of cassette recorder. One of the portable mains/battery types which has an automatic recording level, a tape counter and remote motor control is ideal. I use a Curry's Trophy CR100, which is excellent, but there are several models by Ferguson, Sanyo and other manufacturers in the £20 to £30 price range which are equally suitable. You will also need a cable to connect your recorder to the ORIC, and this is a rather specialised piece of equipment. It uses a seven-pin plug (Fig. 1.1) of the type

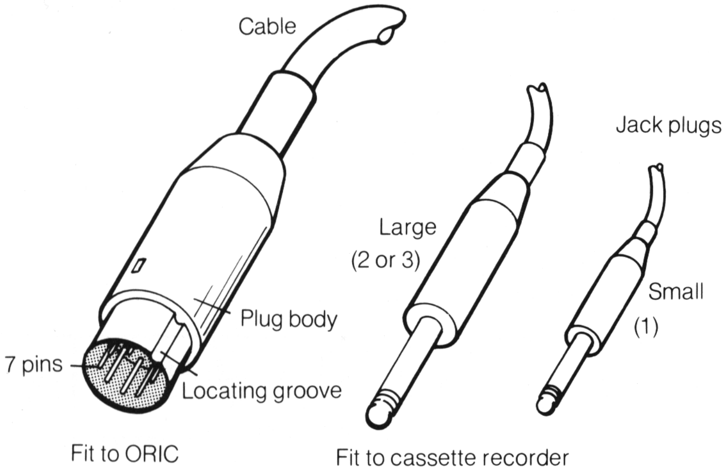


Fig. 1.1. The 7-pin DIN plug and the jack plugs of the cassette cable. Production ORIC cables may use a fourth plug for hi-fi amplifier input.

called a 'DIN-plug' at one end and three or four connectors at the other end. A cassette cable for the BBC Micro will fit, but for getting the most out of your ORIC you need its own cable. By the time you read this, the cable may be supplied along with the ORIC. I used a very early machine, and I had to make my own cable. A full-scale cable for the ORIC should have four plugs fitted at the end remote from the computer. Of these, three are for the cassette recorder, and one is for connecting the sound signals from the ORIC to a hi-fi system so that you can make full use of the ORIC's excellent sound synthesiser. More of that later, though.

Power to the ORIC

Connecting power to the ORIC is the easiest task in computing! The plug is already connected, and the transformer which converts mains voltage down to the safe low voltage that the ORIC uses is built into the plug. This makes the plug very large, however, and you may have problems when you want to connect the mains plugs for the ORIC, the cassette recorder and the TV all to one socket. The conventional type of adaptor will not take the ORIC combined plug/transformer happily, and by far the best solution is to buy one of the four-way socket strips that are sold in all the electrical shops. If you can buy one which has a length of wire and a plug already attached, this will save you a lot of effort. A 5A fuse is all that you will need in the plug for the socket strip.

The ORIC power plug/transformer sends its output along a length of thin twin-wire to a small hollow plug. This fits into the power socket on the ORIC, which is at the right hand side as viewed from the back (Fig. 1.2). There is no switch on the ORIC, so it's tempting to switch on and off by pushing in and pulling out this plug. Don't be tempted to do this – if you do so it will become loose

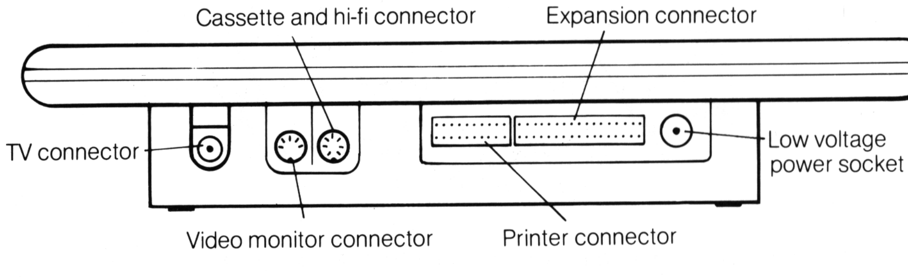


Fig. 1.2. The connectors on the back of the ORIC.

and your ORIC will not work reliably. Do all of your switching on and off by using the mains plug or by switching the mains socket on or off. This is safer and won't lead to any problems with connections.

TV connection

When you have dealt with the mains supply, you should have a working ORIC, but you need now to check this. You need to connect the ORIC to a TV receiver to see what the effect of a working ORIC is. The TV connector cable is illustrated in Fig. 1.3 –

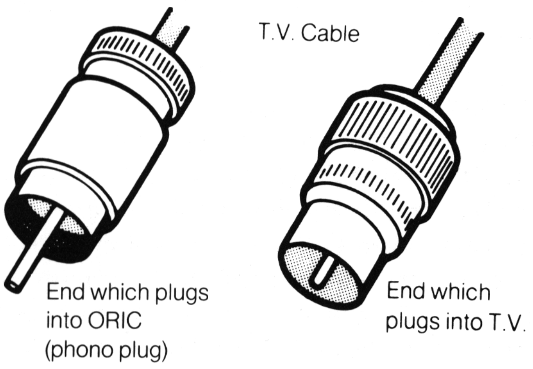

T.V. Cable
End which plugs into ORIC (phono plug)
End which plugs into T.V.

Fig. 1.3. The TV connector cable and its two plugs.

the plugs at the two ends are differently shaped. The plug with the protruding contact fits the socket on the ORIC, the other one fits the aerial socket of the TV. If you want to keep the TV in normal use as

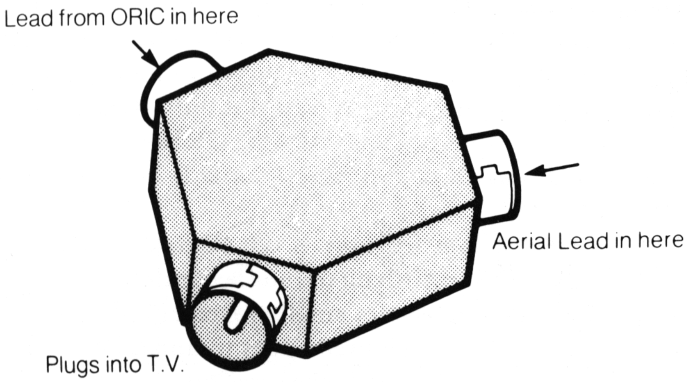


Fig. 1.4. A typical TV two-way adaptor, useful if you have to share your ORIC with Coronation Street.

well, buy one of the two-way adaptors (Fig. 1.4) which is sold under various brand names.

Having connected up so far, you can try it all out. Switch on the TV and turn down the volume control. Switch on the ORIC. The ORIC is now transmitting a signal to the TV, but the TV will not, unless you are lucky, be tuned to this signal. Before you can see the picture that the ORIC produces on the screen, you must tune the TV to the ORIC signal. Fig. 1.5 shows the methods that are typically used to tune TV receivers. The simplest method is the dial (Fig. 1.5(a)) that is used by black-and-white portables. Such a portable is perfectly suitable for using with the ORIC. You can't see the colours that the ORIC produces, obviously, but you will see shades of grey, and all of the other features will be displayed just as well as they would be by a colour receiver. The normal ORIC picture is black on white, in any case.

Other tuning methods are used for colour receivers. When a receiver uses large push-buttons (Fig. 1.5(b)), tuning is carried out by pushing a button fully in, so that it locks, and then turning the button. You should use a button that you don't normally use for a TV station, and you have to keep turning it until the signal from the ORIC appears. If you find that you can turn the button through several revolutions without finding the ORIC signal, then keep turning until it will turn no more. If you still can't find the signal, then turn in the other direction. At some position you will see appearing on the screen the ORIC 'signature', as shown in Fig. 1.6.

More recent designs of colour TVs use touch-buttons or pads to select different stations. Touch one that is not normally used for TV. You will generally find six or twelve such buttons, so use the last one to avoid any confusion with TV station settings. Now you have to look for a small flap which lifts off to reveal a set of tuning wheels (Fig. 1.5(c)). Find the one which tunes the button that you have selected (they are numbered) and turn it until the ORIC 'signature' appears. As before, if you don't find the ORIC signature by turning all the way in one direction, then you will have to try the other direction.

What you must aim for now is the clearest possible picture. A colour TV is very often difficult to tune accurately, and you will need a fairly delicate touch. A very small movement of the tuning wheel, for example, will cause coloured edges to appear and disappear on letters. Try to aim for letters on the screen that are free of the defects illustrated in Fig. 1.7.

You should be just about out of the wood as far as the TV display

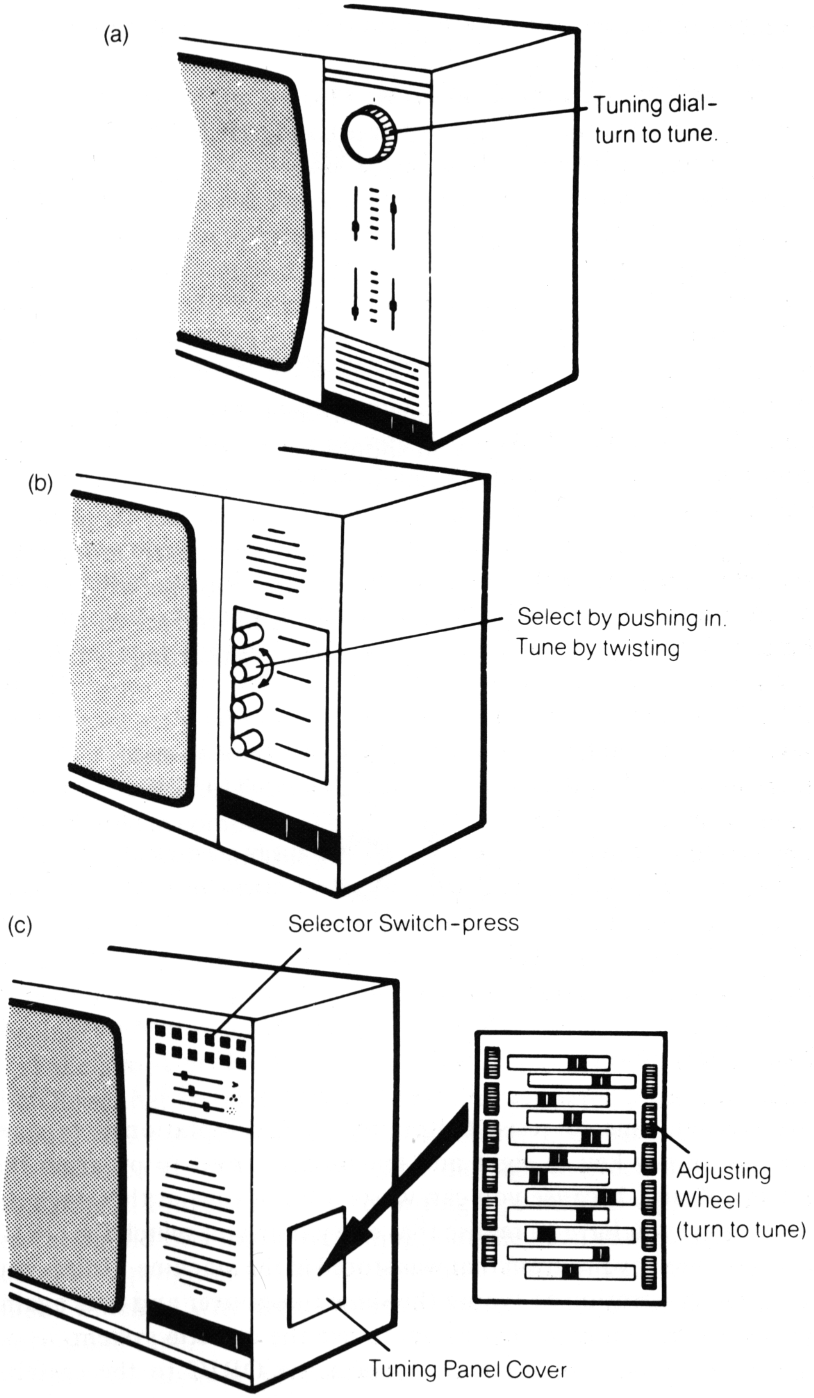
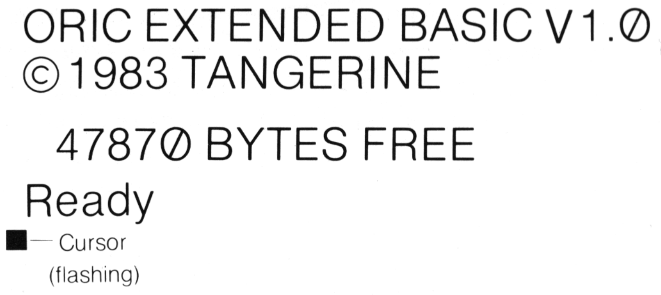


Fig. 1.5. TV tuning methods: (a) dial tuning, (b) large pushbuttons, (c) small pushbuttons with a tuning panel.



```
ORIC EXTENDED BASIC V1.Ø
©1983 TANGERINE
  4787Ø BYTES FREE
Ready
■
```

Fig. 1.6. The ORIC 'signature' on the screen.

is concerned now, but you may find, as I did, that the picture sometimes 'rolls' vertically. You should be able to keep the picture steady by using the 'vertical hold' control on older TV designs. If your TV does not have such a control, but gives perfect TV pictures, then the fault may be in the ORIC. Mine was a *very* early model, and the later production models should give no problems of this type. Don't in any case attempt to make any adjustments to the TV, particularly inside adjustments, unless you are thoroughly familiar with TV circuits.

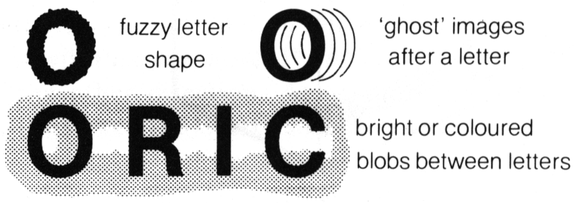

fuzzy letter shape
'ghost' images after a letter
ORIC
bright or coloured blobs between letters

Fig. 1.7. Some defects caused by faulty tuning.

Cassette capers

Now is the time to resist temptation. The temptation is to start pressing every key in sight and teaching yourself to program the ORIC. Resist, because you can waste a lot of time at this stage. In particular, you have to realise that every time you switch the ORIC off, you lose any program that was stored in the machine. Unless you want the frustration of typing the same things over and over again, you must at this stage learn how to use the cassette system.

To start with, you need to connect the ORIC to the cassette recorder. The connection at the ORIC uses the seven-pin plug which has to be the correct way up to be inserted. Push this plug gently into place, and then connect the plugs, called jack plugs, into the cassette recorder. The very small plug controls the motor of the recorder, and it plugs into the socket that is usually marked REM, meaning REMote control. If your cassette recorder does not have such a socket, don't worry – you can still use the recorder. You will, however, have to use the recorder more carefully when we come to use it for data recording in Chapter 5.

The other two recorder plugs are more important. One of them plugs into the MIC (microphone) socket of the recorder, the other one into the EAR (earphone) socket. The EAR socket is sometimes marked with a small drawing of an ear rather than with the word. The problem now is – which plug goes into which socket? My early ORIC did not come with cassette leads, so I had to make do with another set from another machine. This particular set used colour coding – the plug which fitted into the EAR socket was grey, the MIC plug was black. If your leads have no form of marking you will have to find out for yourself. Fortunately, this is not difficult, and the method of finding out is much the same as the method of checking that the cassette recording is going according to plan. Here's how.

To start with, you need a program and a blank cassette. For this use it's a waste of money buying C-90 cassettes – use the short-length computer cassettes (such as C-10, C-15) that you can find in W.H. Smith's and in computer shops. You can also be sure that the tape on these cassettes is suitable for computer use. Avoid at all costs the 'special offer Super-Fi' tapes from manufacturers you have never heard of. I use C-90 tapes for working on very long programs, and I have found tapes by Sony, Agfa, Scotch, BASF, Maxell and Yashima to be acceptable. You don't need the CrO_2 types, just the ordinary LN (low noise) tapes.

Take a look at the cassette. When the tape is wound to one end, you'll see that the end is made of clear or coloured plastic. This is a 'leader'. It's not recording tape, and you can't record anything on it. It's there to ensure that you can't damage the recording tape by touching it. Finger prints on recording tape make a slight difference to the sound of a music recording, but they can cause disaster to a computer program stored on the tape. The same is true of video tape, which is why video cassettes have a flap which completely covers the tape.

Put the cassette in the recorder, with Side 1 (or Side A) uppermost, and reset the tape counter to zero. Then fastwind the tape until the tape counter reads 5. If you have no tape counter, then, with the cassette out of the recorder, push the body of a BiC pen into the centre of the empty reel of the cassette and wind the tape by hand until you see the brown recording tape take the place of the plastic leader – then wind on a few turns more. The cassette is then ready for use. What we need now is something to record.

We need a program, and that means typing. Start by taking a look at the keyboard. In particular, find the rather larger key on the right hand side that is labelled RETURN. That is one of the most important keys on the computer, because pressing the RETURN key is a signal to the computer that you want it to carry out some action. Press the key. You may see a message appear on the screen if you have previously been pressing other keys, but don't worry about that.

Now type 1∅ – press the 1 and then the ∅. Be careful here – the ∅ has a slash mark across it so that you don't confuse it with the letter O. If you make a mistake, press the DEL (delete) key once. Don't keep any of the keys pressed down, because their actions will repeat for as long as they are held down, giving you 1111∅∅∅∅∅∅ instead of 1∅! Following the 1∅, type the letters REM. It doesn't matter whether you leave a space between the ∅ and the R or not. REM means REMinder to the computer and it is one of the 'keywords' that the computer will recognise. REM means that whatever follows is a reminder to the reader, not an instruction for the computer. We use REM to write reminders for ourselves in programs. If we typed:

```
1∅ THIS IS A TEST
```

as part of a program, the computer would object when we wanted it to carry out the program instructions, because it looks for an instruction word, and THIS is not one of the ORIC 'keywords'. You will find REM used throughout this book as a brief reminder of what each program example does.

Now when you have typed 1∅ REM, press the RETURN key. The brief bleep that you hear when you press this key is at a lower pitch than the one you hear when you press the other keys. This is a useful reminder. The computer prints its 'cursor' now on the next line down. The cursor is the flashing black square which shows you whereabout on the screen the next typed letter will appear. Once you have pressed RETURN, the computer has placed a coded version of what you typed into its memory and is waiting for another instruction. Now type 2∅ REM and press the RETURN key again. Now type 3∅ REM and press RETURN; follow this up with 4∅ REM and press RETURN again.

Now this may not look very much, but it's a program of sorts and we can record it (Fig. 1.8). Just to prove that the computer has memorised the program, type LIST and press RETURN again. Your typing appears again, showing that you can display what the computer has memorised at any time.

```
1Ø REM
2Ø REM
3Ø REM
4Ø REM
```

Fig. 1.8. A simple program for test purposes.

We can now record the program. Make sure that the cassette is ready for use, and type:

```
CSAVE"T",S
```

CSAVE means Cassette Save – save the program on a cassette. The letter T is a 'filename', a kind of label which will also be recorded and which the ORIC can use to recognise this program again. We could use "TEST" in place of "T" if we wanted, but I'm trying to reduce the amount of typing that you have to do. The inverted commas or 'quotes' have to be placed on each side of the 'filename', otherwise the computer will not obey. Computers are not intelligent – they can only do exactly as they have been instructed – and the computer will recognise T as a filename only if it is enclosed in quotes and immediately following the CSAVE. Incidentally, the filename "T" is not taken as being identical to " T" or "T ", because the computer recognises the space as being part of the filename!

The S means Slow. The ORIC is arranged so that it can record and replay at two speeds. That doesn't mean that the motor of the recorder moves any faster or slower, just that the rate of sending signals to the tape or receiving them back can be changed. The slow speed of the ORIC is very reliable, and you should always use it for programs that you value. If you omit the comma and the S, the recording will be at a very high speed – but more about this later.

The screen should now show:

```
CSAVE"T",S
```

and you can press the RECORD and PLAY keys of the cassette recorder. This starts the tape moving if you have no remote motor control, but if you have inserted the remote control plug, the tape will not move. When you press the RETURN key of the ORIC, the tape will start to move if the motor control is fitted, and the program will be recorded. When the actual signals are sent out, the words SAVING T will appear at the top of the screen. When the program has been saved, these words will disappear, and the signal word, Ready, will appear on the screen. The motor will stop if remote control is fitted, and you should also use the STOP key on the recorder to release the mechanism.

These steps should have recorded our simple program, but we need to be sure of this, particularly if we are not certain which plug fits the MIC socket and which fits the EAR socket. If the plugs are the wrong way round there will be no true signal on the tape, so the first thing to try is to find if a signal has been recorded. Rewind the tape, make sure that no plug is inserted in the EAR socket, and set the volume control of the recorder to halfway. If there is a tone control, set it to the position of maximum treble. Now replay the tape, ignoring the computer, and just listen to it. If there is a loud steady tone, followed by a brief warbling note, then this is the correct recorded signal, and you have the correct plug in the MIC socket. If you find no signal, or only a very faint signal on the tape, then try again with the other recorder plug in the MIC socket.

Once you are certain that you have achieved a recording, you can check the replay or 'loading' of the program. To be sure, type NEW and then press the RETURN key. This wipes out the program stored in the computer, as you can prove by typing LIST and then pressing RETURN. The blank result of this proves that no program is present. Insert the EAR plug into the cassette recorder. When you do this, you will not hear the signals from the tape.

Now type CLOAD"T",S. CLOAD is the keyword meaning Cassette LOAD (replay the signals from the tape), and we have used the same 'filename' and the S for Slow. Rewind the tape, then press the RETURN key. The word 'Searching' will appear at the top of the TV screen, and you can now start the tape recorder by pressing the PLAY key. Whenever the signals start to reach the ORIC, the word 'Searching' will change to 'Loading', with the filename T displayed. The word 'Ready' will reappear on the screen when the program is back in the memory, and the cassette motor will stop if you are using the remote control. Press the STOP key of the recorder in any case. Now prove that your program is present by typing LIST and then pressing the RETURN key.

On my recorder, the volume control is marked with setting numbers 0 to 10, and ORIC programs would load at any setting between 2 and 8. Very few computers have a tape system that is so tolerant of different settings as this. If you omit the S when you are saving and loading, these operations are carried out at a very high speed, and this higher speed is much more demanding. You will probably find that the volume control of the cassette recorder has to be much more carefully set when you are using the high speed than is necessary for the slow speed.

You can now try to load the ORIC test/ demonstration tape. The sample which I had was recorded at slow speed, and could be loaded by typing:

```
LOAD"",S
```

and then pressing RETURN. The use of "" as a filename will load any program; you don't then have to know the correct filename. This is useful if you are not sure of a filename, because unless you get the filename exactly right, the computer will ignore it. If you find that the demonstration tape will not load, but that the computer works with your own tape, then your head needs looking at – the cassette-recorder head, that is! See Appendix A for details of how to carry out this adjustment, which is surprisingly often necessary.

Keyboard topics

Before we go any further at the moment, take a close look at the layout of the keyboard of the ORIC. The keys are small, and have a click action, but the spacing of the keys is the same as that of a typewriter. You will find that the layout of most of the keys is also very similar to the layout of a typewriter, but with a few exceptions. One exception is that the quote marks " and ' are on one key next to RETURN. The main differences, however, are the RETURN key itself, and the ESC and CTRL keys. The RETURN key, as we have seen, is a signal to the computer that you want it to carry out an action. On the typewriter, this key would cause a 'carriage-return', so that the typewriter selected the next line and returned to the left-hand edge of the paper. You *do not* have to press the RETURN key of the ORIC when you reach the right-hand side of the screen with letters. The computer will select a new line automatically, and you use RETURN only when you have finished typing something and you want the computer to act on it.

The other two of these special keys, ESC and CTRL are keys which perform special actions when they are pressed at the same time as other keys. The CTRL key will carry out some action when it and another key are pressed together. The ESC key will carry out some special action when it is pressed, released, and then one of the other keys is pressed. We will see how these keys are used later, but for the moment, it's useful to know that you can clear the screen by pressing CTRL and L together. We shall represent this on paper as CTRL L. When you clear the screen in this way, you do not clear the memory of the computer.

The SHIFT keys operate like typewriter shifts, but you won't have had much chance to see this. A typewriter normally operates in *lower case* (meaning small letters) and obtains *upper case* (capitals) by pressing SHIFT along with the letter key. The ORIC will accept instructions only when they are in upper case, so when you first switch on, this is all you can type. If you want to be able to use the keyboard like a typewriter, press CTRL T (CTRL and T together). Now when you press a letter key you will get lower case, and when you press SHIFT with a letter key you will get upper case, just like a typewriter. You can go back to normal typing in upper case by pressing CTRL T again. You will still have to press the SHIFT key to obtain the double quotes and the symbols above the number keys.

Now we are set to start on the way to programming ORIC so as to provide you with all the computing power you will need for a long time to come.

Chapter Two
Numbers and Strings

Everyone expects a computer to be able to work with numbers, but the methods that the computer uses are not the same as those of a calculator. A calculator will display a number when you press a key. The computer will also do this, but to carry out arithmetic needs more than just the use of the 'equals' sign. If you type 5 − 2 = on your computer, no answer appears. That's because a computer is a machine, with no brain. It can carry out a set of commands, even a very long and complicated set of commands, but only the commands that have been built into it. There's nothing new about this – washing machines make lousy central heating boilers!

If you want the result of a piece of arithmetic to appear on the screen, you have to make use of the instruction word PRINT. PRINT means 'print-on-the-screen', not print-on-paper, and if you type PRINT 5 − 2 then you are commanding the computer to work out and then display this subtraction. It doesn't, though! The reason is that the computer does not carry out a direct command like this until you press a 'make-it-go' key, the RETURN key. If you have just typed PRINT 5 − 3, press RETURN and you will see the answer 2 appear under the command PRINT 5 − 3.

Why do we need this extra step? Simply because it gives you time to check what you have typed and to add to it if needed. If you had typed PRINY or PRIT or used the wrong sign in place of the subtract sign, then the computer could not proceed. It would indicate this by the message SYNTAX ERROR. Syntax error means a mistake in the way a command is typed. This could be a spelling error in a command word, an incorrect sign, or a command word used in an incorrect way. Once again, the computer cannot think for itself – it obeys only the commands that are built into it, and only in one precise form of each. That's what we mean by syntax. If you get the command 100% correct, that's just about good enough!

Having to use the RETURN key to make the computer do anything, then, gives you time to sort out your mistakes. We'll look in detail at sorting out errors in Appendix B, but for the moment, the simple way of correcting mistakes before you press RETURN is the use of the key marked DEL. DEL means DELete, and when you press this key, the cursor moves left and leaves a blank space where there was a character (letter or digit) before. The cursor, remember, is the marker that appears on the screen which shows where the next character will be typed. As you move it to the left it rubs out as it goes. You don't have to tap the delete key five times to move it five spaces to the left, either. When you hold down the DEL key, the action repeats, and you can watch the cursor fly off to the left, gobbling up characters as it goes. Don't get too bemused by this, or you will wipe out the whole line that you typed!

You can carry out arithmetic, using the + sign for add and the − for subtract, but the multiply sign is *, not ×. The ORIC-1, like all computers, is arranged to 'recognise' × as a letter only. The division sign is /, but otherwise there are no great surprises for you if you have ever used a calculator. The two main points to remember are that you have to start with PRINT if you want to see an answer, and that you have to press RETURN to make it all happen.

Now try this. Clear the screen by pressing CTRL L (the CTRL key and the L key together). Now type:

```
PRINT"CONSULT THE ORIC"
```

Note the use of quotes (inverted commas). When you press RETURN now, what you will see appear on the screen under your line of typing will be the words CONSULT THE ORIC. The quotes don't appear. That's because they are part of the PRINT command, which doesn't appear either. The effect of using quotes is that the computer prints whatever is placed between the quotes, just as you type the words, no matter how you spell them. Now to see a very important difference, type:

```
PRINT"2*3= ";2*3
```

and when you are sure that it's correct (watch for the space between the = and the "), press RETURN. What appears on the screen now is:

```
2*3= 6
```

The piece that was enclosed in quotes is printed exactly as it is. The piece outside the quotes, following the second quote mark, is worked out and the *result* is printed. The semicolon is there as a signal to the computer that there is more to print – we'll look at that point again later. You can, of course, cheat! If you type:

```
PRINT"2*5= ";2*3
```

then, when you press RETURN, the computer will come back with 2*5= 6. I told you it didn't think for itself!

Whatever is placed between quotes is printed, any arithmetic commands that are not placed between quotes are worked out and the result printed. If you type something like PRINT A (press RETURN), however, you get a blank line! There's a good reason for this, and we'll come to it later under the heading of variables.

Commands and instructions

So far we have been using *commands*. A command is something that the computer has to carry out (if it can) directly when the RETURN key is pressed. That's not much more than a calculator can do, and the real advantages of using a computer emerge when you program it. Programming means that you issue a set of instructions in a sequence. The computer can then be commanded to carry out the sequence of instructions in the order that you have decided. This is where real computing starts!

Throughout this book, we'll use the word command to mean a direct command – you type it, press RETURN and it's done. Anything that appears in a program, though, we shall call an *instruction*. The difference is that a program instruction is not carried out when you press RETURN, only when you have finished typing all of the instructions and you want them all to be carried out. The computer needs some way of distinguishing these instructions from commands, and the way that we use is a *line number*. A line number is a positive whole number, anything between ∅ and 32767. If the first item that you type immediately after switching on, or after pressing RETURN is a number like this, the computer will take everything that follows as being program instructions until you press RETURN again. You can't use fractions or negative numbers as line numbers. You could number your lines 1,2,3 ... but we normally use 1∅, 2∅, 3∅ ... for good reasons that we'll look at later.

Try the very simple program in Fig. 2.1. Start by clearing the screen (press CTRL L) and then type the first line number 1∅. The number keys are on the top line of keys, remember, and the zero has

```
10 REM PRINT
20 PRINT"ORIC-1"
30 PRINT"IS"
40 PRINT"TOPS!"
50 END
```

Fig. 2.1. A simple PRINT program.

a slash through it so that you don't confuse it with the letter O. When you are satisfied that a line is correct press RETURN, and then type the number that will signal the start of the next instruction line.

Notice the difference? When you press RETURN, nothing is carried out; the cursor simply moves to the next line. These are program instructions (or *statements*) rather than direct commands. When you use line numbers and then type program instructions, the computer stores these line numbers and instructions in its memory, ready to carry out later. This isn't done for a direct command – when you press RETURN, a direct command (which is memorised up to the time you press RETURN) is carried out, and the memory is cleared again. The only way you can repeat the action of a direct command is by typing it out all over again and pressing RETURN.

How can we prove that this program is stored in the memory? It's easy – type the word LIST and then press RETURN. LIST, you remember, is a command word meaning 'print a list of the program on the screen'. Unless you switched off the computer, which always clears the memory, you will see the program lines appear when you press RETURN. Now to make the program happen, type RUN and press RETURN. RUN is a direct command which makes a stored program work. Take a look at what appears on the screen.

Your program has been run, and the instructions have been carried out. They have been carried out in the order of the line numbers, starting with 1∅ and ending with 5∅. What's more, the computer will keep the lines in the correct order. Try this – type:

```
15 PRINT"THE"
```

and then press RETURN. Now type LIST and press RETURN. You will see that line 15 has been put into its correct place between lines 1∅ and 2∅. If you type RUN and press RETURN now, you will see that line 15 is obeyed in the correct sequence, putting the word THE into place.

Two puzzles remain. First, what did line 1∅ do? The answer is nothing! It gets printed when you LIST your program, but the computer ignores it when you type RUN (press RETURN). This is because, as we saw in Chapter 1, REM means reminder – just a way of reminding you when you LIST a program of what it is supposed to do. It isn't something that you want the computer to carry out, just to store. You can type whatever you want following REM – it will be printed in a listing, but otherwise ignored by the computer. Line 5∅ contains the new instruction END. This indicates to the computer that there are no more instructions, the program has ended. Strictly speaking, you don't need to use this instruction when there are no more lines following line 4∅, the last instruction line, but using END is a good habit to cultivate.

Now try something different. Clear the program out of memory by typing NEW and then pressing RETURN. Clear the screen (remember how?). These actions will remove the clutter on the screen and make way for a new program. The words on the screen move up and disappear off the top of the screen when you have 27 lines of words. This action is called *scrolling*, and it means that each new line will from then on appear at the bottom of the screen. Clearing the screen allows you to see your typing appear at the top of the screen again, and clearing the memory makes sure that a new program is not ruined by lines left over from an older one.

Now type the three instruction lines of Fig. 2.2. Lines 1∅ and 3∅

```
 1∅ REM LONG LINE
 2∅ PRINT"THIS IS A MUCH LONGER LINE T
HAN YOU HAVE TYPED BEFORE"
 3∅ END
```

Fig. 2.2. Typing a long line. Do NOT use RETURN in line 2∅ until you have typed the final quote mark.

are straightforward enough, but line 2∅ is a long line. The point here is that you keep typing letters until you reach the final quote mark – *then* you press RETURN. Don't be tempted to press RETURN after THAN, because when you press RETURN the computer takes this as meaning the end of your typing of line 2∅. Don't worry about where the letters appear. You will see that the computer organises things so that the second part of the line appears under the first part on the screen.

All of this is intended to drive home the point that an instruction line is not just one line on the screen. An instruction line on the ORIC can be up to 80 characters (80 presses of keys), which is two lines on the screen plus a few characters to allow for the spaces that appear in front of the line numbers. If you get near to the 80 character limit, a 'ping' will sound to remind you, and when you see a slashmark (/) after a character you have exceeded the limit, and the line will not be accepted. When this happens, you have lost the line completely – the DEL key will not space back over the line because the cursor will by now be on the next line. When you hear the 'ping' therefore, stop typing, and delete if you are in the middle of a word. If you want to hear what the 'ping' sounds like, just type PING and press RETURN.

Printing-plus

When we use the instruction PRINT, the computer has a set way of carrying out this instruction. Each item (between quotes) that follows the PRINT is placed on a new line of the screen, scrolling if necessary, and at the left-hand side. This isn't always convenient for our purposes, and ORIC allows us to change this in several ways. Clear out the memory and the screen, and try the program in Fig. 2.3. Watch out, as you type, for two things. One of them is spaces –

```
10 REM SEMICOLONS
20 PRINT"ORIC-1 ";
30 PRINT"IS ";
40 PRINT"TOPS!"
50 END
```

Fig. 2.3. The effect of semicolons. Don't forget the spaces in lines 2∅ and 3∅.

there is a space between the last letter of the word and each quote mark that follows it. The other thing is where the semicolons are placed in lines 2∅ and 3∅. The semicolons come last, after the quote marks. This way they are not printed along with the words, instead they act as a signal to the computer. When you RUN this program, you will see all the words appear on one line. If you see words joined together it will be because you have missed out a space somewhere. We'll look later at how to edit (alter) a line (see Appendix B) but for the moment, if you want to change a line, just type its line number and then the correct version of the line and press RETURN to fix this new version in the memory.

Another way of altering the way that items can be printed is the use of commas. Try the program in Fig. 2.4, and see what happens when this runs. The comma is a 'field' instruction. It causes items to be placed with each item at the start of an invisible column on the screen, with three columns per screen width. Because of this, the first

```
10 REM COMMAS
20 PRINT"A","B","C"
30 PRINT 5,10,15
40 PRINT"E","F","G"
50 END
```

Fig. 2.4. The effect of commas.

item on a line is put in column 1, the second in column 2 and the third in column 3. Note that letters have to be enclosed in quotes, but numbers don't. The commas in lines 2ϕ and 4ϕ must be placed *outside* the quotes. If you type "A," then the comma is printed along with the A and there is no spacing effect. The screen permits a maximum of 40 characters across its width. If you try to use more than three columns, the additional items will appear on the next line down. An overflow from one column to the next will be caused if you make an item in any column longer than the space between one column and the next. Fig. 2.5 illustrates these points. On my early model of ORIC, numbers were not placed in the same position as letters in each column – this may have changed by the time you read this.

```
10 REM OVERFLOW
20 PRINT"A","B","C"
30 PRINT"THIS IS TOO LONG","TO","FIT"
40 END
```

Fig. 2.5. Overflowing columns. If you have more items than columns in a line, then a new line is taken. If an item will not fit within a column, one or more columns will be skipped.

Tabulation

Commas are useful if we want to divide the screen into set columns. A much more flexible arrangement, however, is called *tabulation.* Tabulation means that you can choose the position along a line where printing will start. The instruction word which carries this out is TAB, and TAB must always be followed by a number enclosed in brackets. The number must be between ϕ and 39, and it is this number which specifies where the printing will start. ϕ means the left-hand side of the screen, and 39 is the right-hand side – the 40-character width of the screen is numbered ϕ to 39, rather than 1 to 4ϕ. We can't use TAB by itself; it must always follow PRINT.

Try the example in Fig. 2.6. Line 2ϕ prints the word ORIC-1 so that it is centred on its line. Line 4ϕ prints the rest of the message at

```
 10 REM TAB
 20 PRINTTAB(17)"ORIC-1"
 30 PRINT
 40 PRINTTAB(4)"COMPUTERS";TAB(20)"RULE"
;TAB(33)"O.K."
 50 END
```

Fig. 2.6. The tabulation instruction, TAB.

different places controlled by the numbers that are placed between the brackets of the TAB instructions. Note that more than one TAB can follow a PRINT. Fig. 2.7 shows the method of calculating the TAB number to use so as to centre a word on a line – it's a method that typists have used for generations!

(a) Count the number of characters (including spaces) in the title.
(b) Divide the result by two.
(c) Subtract from 20 and use the final result as the TAB number.

Fig. 2.7. Calculating the TAB numbers for centring a title.

Tabulation isn't the only way of spacing words or numbers from each other. There's another instruction word SPC, which means SPaCe. Like TAB, SPC has to be followed by a number enclosed in brackets. The number does not specify a position measured from the left-hand side of the screen, however. Instead, the number is measured from the last character of the previous word or number – it's the number of spaces that you want to put between one character and the next. Try the example in Fig. 2.8 to see the difference.

```
 10 REM SPC
 20 PRINT"ORIC-1";SPC(3)"IS THE";SPC(3)"
BEST";SPC(3)"VALUE"
 30 END
```

Fig. 2.8. Comparing the effects of SPC and TAB.

These examples will have illustrated another important point. Provided that you start the instruction with PRINT, you can use TAB and SPACE more than once and mix them together. There are some rules, though. You can't tabulate to the left. If you use PRINTTAB(1∅)"A", you can't then use ;TAB(6)"B" in the same line and expect it to print to the left of the "A". Once the cursor has got across to position 1∅, it will not go back to position 6. The other point is that you have the option of using semicolons. You can type PRINTTAB(2)"A"TAB(1∅)"B"TAB(15)"C", with no semicolons, or you can type PRINTTAB(2);"A";TAB(1∅);"B";TAB(15);"C", with semicolons. The ORIC, unlike some of its rivals, will accept either version. Personally, I use the semicolons to make it easier to see what I am printing. Note incidentally that you can use PRINT TAB or PRINTTAB ; the computer accepts either version.

Point of entry

So far, we've used the ORIC only for outputs, printing on the screen only what we have commanded by instructions in the program. This is one main way in which the ORIC can show you what it is doing. The other is by printing on paper. There are, however, ways in which we can pass information to the ORIC so that it does not have to be put into a program beforehand. Suppose we wanted to instruct the ORIC to print your name. Now I don't know your name, so I can't write a program that has your name in it. I can, however, get the ORIC to print your name with a little help from you. This bit of magic is illustrated in Fig. 2.9. When you run

```
10 REM INPUT 1
20 PRINT"PLEASE TYPE YOUR NAME NOW"
30 INPUT NM$
40 CLS
50 PRINT NM$;" -THIS IS YOUR LIFE!"
60 END
```

Fig. 2.9. The INPUT instruction – the computer waits for you. Remember to press the RETURN key when you have typed your answer.

this program, the computer prints the request in line 2∅, and then stops. The INPUT instruction in line 3∅ is what makes it stop and wait. It is waiting for you to type something, your name, and then to signal that you have done so by pressing RETURN. When you do so, you will see your name appear in the famous phrase in line 5∅.

Now how does this happen? The PRINT instruction in line 5∅ doesn't contain your name – or does it? The clue is the use of NM$. That's not an 'S' at the end, it's a dollar sign, but NM$ is pronounced 'en-em-string' rather than 'en-em-dollar'. A *string* means a set of letters or other characters, and we use NM as a 'code' name or, more correctly, a *variable* name.

NM$ doesn't look much like your name. What has happened here is that line 3∅ *assigns* your own name to this code. Wherever NM$ is used from then on, your name will be substituted. This way, I can put in instructions to print whatever you typed in line 3∅ without having to know in advance what it is. You don't need to put any quotes around your name when you type it, and you don't need to put any quotes around NM$ when you type that. Using these codes or variables gives us a lot more freedom in the way that we design and use programs. Why call them variables? It's because we can vary what we want the code to represent at any point in the program. Until we change it, though, it stays assigned.

We can assign a name to a variable without using INPUT. Take a look at Fig. 2.10. In line 2Ø, the instruction MM$ = "Mickey Mouse" means that whenever we use MM$ from that point

```
10 REM ASSIGN
20 MM$="Mickey Mouse"
30 PRINT"PLEASE TYPE YOUR NAME"
40 INPUT YN$
50 CLS:PRINT
60 PRINT"ARE YOU SURE- ";YN$;" ?"
70 PRINT"YOU'RE NOT ";MM$;" ?"
80 END
```

Fig. 2.10. Assigning variables. Some computers require LET before MM$ in line 2Ø.

onward, the computer will take that as meaning Mickey Mouse. Note that we *do* need the quotes around the name when we assign it this way. A very few computers force you to type this as LET MM$ = "Mickey Mouse", but ORIC follows the Microsoft standard of BASIC which doesn't need the word LET. But, if you *do* type LET, ORIC will not object.

Line 4Ø assigns your name to YN$ (no quotes needed), and lines 6Ø and 7Ø do the printing of the names and some comments. Just one more point about this example: CLS means 'clear the screen'. It's the same action as you carry out as a direct command by using CTRL L. By putting this instruction into the program we ensure that the printing will be on a blank screen with nothing to distract you. From now on, we'll use the CLS instruction at the start of most of our programs.

This very convenient use of variables as codes applies to numbers as well. A number variable name can be a letter, two letters, or a letter and a number. There's no dollar sign, though, because it's the dollar or string sign that distinguishes a string from a number in a variable name. Take a look at Fig. 2.11. We've used the variable name of N in line 4Ø, so when the program hangs up waiting for you to type the number, type a number between 1 and 1Ø as requested, and press RETURN to watch the result.

```
 5 REM NUMBER VARIABLE
 10 CLS
 20 PRINT"PLEASE TYPE A NUMBER"
 30 PRINT"MAKE IT BETWEEN 1 AND 10,PLEAS
E"
 40 INPUT N
 50 PRINT"3.6 TIMES ";N;" IS ";3.6*N
 60 PRINTN;" DIVIDED BY 4 IS ";N/4
 70 END
```

Fig. 2.11. Using number variables.

Fig. 2.12 illustrates the point about a variable name being used more than once, hence the name of variables. In line 2∅, A is assigned to the number 15, and line 3∅ prints this value. In line 4∅, A is assigned to the number 24, and the instruction in line 5∅ is exactly the same as the instruction in line 3∅ – but it prints 24 this time. Similarly, line 6∅ makes A equal to 32, and line 7∅ prints this value. Just to cap it all, line 8∅ asks you to type a number for yourself and line 9∅ prints it. Once again, the printing instruction is the same, just PRINT A. It makes you wonder if you need to type this instruction so many times ... but that's jumping the gun on the next chapter!

```
5 REM VARY IT!
10 CLS
20 A=15
30 PRINT A
40 A=24
50 PRINT A
60 A=32
70 PRINT A
80 INPUT"Your favourite number";A
90 PRINT A
100 END
```

Fig. 2.12. Varying variable values – hence the name!

Chapter Three
Disorderly Behaviour

So far, the programs that we have looked at have been the type that we call *linear programs.* Linear means that they start at the lowest line number, carry out each line in turn, and end at the highest line number, always taking the same path and carrying out the same lines. We can write programs which are not linear, but only if we have an instruction that allows the computer to decide which line to carry out next.

```
 5 REM DECISION
 10 CLS:PRINT
 20 PRINT"TYPE Y OR N"
 30 INPUT A$
 40 IF A$="Y"THEN PRINT"THAT MEANS YES":
GOTO 70
 50 IF A$="N"THEN PRINT"THAT MEANS NO":G
OTO70
 60 PRINT"YOU CHEATED!"
 70 END
```

Fig. 3.1. A set of decision steps using IF...THEN. Notice the use of a second instruction in lines 4ø and 5ø, separated by a colon.

The keyword for this type of decision is IF. Take a look at Fig. 3.1. Line 2ø asks you to type Y or N, and line 3ø waits for an input. Whatever letter you type is coded as variable A$ and is then used in lines 4ø and 5ø. In line 4ø, A$ is compared to Y (note the quotes around the Y in the program). If the two are identical, which means that you pressed the Y key, then the program prints THAT MEANS YES and the last instruction in the line, GOTO 7ø, ends the program in line 7ø. GOTO 7ø is a separate instruction, and it has to be separated from the previous one by a colon. A line like this which contains more than one instruction, separated by a colon, is called a *multistatement line.* If you had typed N, the rest of line 4ø would be ignored, and the program would move to line 5ø. If A$ is N, then the printed message is THAT MEANS NO and the second instruction in the line (colon again here) GOTO 7∅ once again causes the program to end. If what you typed was neither Y or N, then lines 4∅ and 5∅ are both skipped, and line 6∅ is the one that is used – YOU CHEATED!

IF has to be followed by what you are testing for. The most straightforward test is equality. Equality (=) as far as the computer is concerned means that the quantities must be *identical.* Almost equal is not enough; the two must be absolutely identical for this test to succeed. When the test succeeds, meaning that the quantities are identical, the actions that follow the word THEN are carried out. If the test fails (the quantities are not identical) then everything that follows THEN is ignored, and the program moves on to the next line. Remember that the equals sign means identical quantities. 3.99999999 will not be taken as identical to 4.00000000, and YES is not the same as yes or Y ES. The computer can't think these things out for itself.

Equality is not the only thing that we can test for, as the table in Fig. 3.2 shows. The marks that are shown here are used in writing mathematical equations, and we also use them in comparisons. Fig. 3.3 shows an example of an inequality. When you put in your number in line 3∅, it is coded as variable N, and in line 4∅, the test is

Symbol	*Meaning*
=	Identity. X = Y means that X is identical to Y.
>	Greater than. X>Y means that X is greater than Y.
<	Less than. X<Y means that X is less than Y.
>=	Greater or equal. X>=Y means that X is greater than Y or equal to it.
<=	Less or equal. X<=Y means that X is less than Y or equal to Y.
AND	More than one test. IF X=∅ AND Y=∅ is a test that is true only if both X and Y are zero.
OR	More than one text. IF X=∅ OR Y=∅ is a test that is true only if either X or Y is zero.
NOT	Tests for opposite. IF X NOT ∅ is true if X has any value that is not zero. This can also be written as: <>

Note: Where two signs are combined, they can be in either order – for example, we can write >= or =>.

Fig. 3.2. The symbols that can be used to cause comparisons, with their meanings.

```
 5 REM COMPARE
 10 CLS:PRINT
 20 PRINT"TYPE A NUMBER BETWEEN 1 AND 10
"
 30 INPUT N
 40 IF N>5 THEN PRINT"IT'S MORE THAN 5":
GOTO70
 50 IF N<5 THEN PRINT"IT'S LESS THAN 5":
GOTO70
 60 PRINT"IT'S 5 EXACTLY!"
 70 END
```

Fig. 3.3. Comparisons of numbers.

to find if this value is greater than 5. If it is, the message is printed and the program stops. If the value is not greater than 5, then the program moves to line 5∅ to check if the value of N is less than 5. If it is, a different message is printed. If the value of N is neither greater than five nor less than five, then it must be 5 exactly, and line 6∅ is carried out.

These two programs have used, unannounced, a new instruction. GOTO means as it says, go to the line whose number is given without carrying out any other lines on the way. This causes the computer to depart from the usual method of carrying out one line after another in strict sequence, and allows us to specify what the sequence will be, depending on the result of the IF test. GOTO can be very useful, but excessive use of GOTO can make the steps of a program very difficult to follow and we have to be careful about its use. Throughout this book, we'll point out the other methods that are available for carrying out the lines of a program out of strict number sequence.

Leaps and loops

A branching program is one that can follow different paths because of a test, like IF A$ = "N". A branch is the name we use when the GOTO is followed by a line number that is greater than the number of the line in which the test is made. If the GOTO is to an earlier line number, this will cause some lines of the program to be carried out more than once. A program that has a branch back like this is called a *looping program*, and the lines that are carried out more than once are the lines of the loop.

Let's look at a simple loop. Fig. 3.4 shows a program that contains a loop. The instruction in line 3∅ is GOTO 2∅, meaning carry out the

```
5 REM ENDLESS
10 CLS
20 PRINT"LOOP"
30 GOTO20
```

Fig. 3.4. An endless loop, repeating line 2Ø.

PRINT instruction over and over again. This is an endless loop, because there is nothing in the program that will stop the loop from repeating lines 2Ø and 3Ø. When this program runs, then, the screen will fill very rapidly with the word LOOP and the screen will scroll from then on. To stop a runaway program like this, you have to press CTRL C (CTRL key and C at the same time). This leaves your program stored in the memory, but stops it. A program that contains an INPUT step is more difficult to stop. There is a small button under the ORIC to deal with such emergencies. The button is recessed so as to make it impossible to press it by accident. I must confess that I found this irritating, having to find a pencil each time I needed to use this button, so I glued a short length of wood dowel to the end of the button, using Superglue. That way, I could push a finger under the ORIC and operate the panic button without having to look for a pencil.

Now that we know how to set up one form of loop, we need to know how to control it so that it doesn't run away like this. A good method is to put a test somewhere in the loop, using an INPUT. Let's see this in action. Fig. 3.5 is a number-totalling program – try it on your cheque-stubs! Line 2Ø sets a variable called N as being equal to zero. This is the variable name that we shall use for keeping the total,

```
5 REM TOTALS
10 CLS:PRINT
20 N=0
30 PRINT"TOTAL SO FAR IS ";N
40 PRINT"PLEASE TYPE A NUMBER"
50 PRINT"TYPING 0 WILL STOP THE PROGRAM
"
60 INPUT J
70 IF J=0 THEN 100
80 N=N+J
90 GOTO30
100 END
```

Fig. 3.5. A running-total program.

and you wouldn't expect the value of the total to be anything other than zero at the start of the program. Line 3Ø prints the value of this total, which will, of course, be zero on the first time round, and then lines 4Ø and 5Ø instruct you on what is to be done. At line 6Ø you get the chance to type in a number (remember to press RETURN). If the number that you type is zero, then the test in line 7∅ will cause the program to end. If the number is not zero, though, a cunning piece of action is carried out in line 8∅. The instruction N = N + J uses the equals sign in a very special way. When the variable on the left hand side is also one that is used on the right hand side of the 'equals', then = means 'becomes'. The new value of N becomes equal to the old value of N (which was ∅) plus the value of J, which is the number that you typed in. In this way, N always holds the value of the total of numbers that you type.

This odd use of the '=' sign can result in instructions which look ridiculous until you are used to them. N = N + 1, for example, means 'add 1 to the value of N', and then make this new quantity equal to N. Another name for this action is *increment N.* The instruction N = N − 1 means make the new value of N one less than the old value; we can call this *decrement N.* N = N*2 means double the value of N and make this the new value of N. If N, or whatever variable name you use, is on one side of the equality sign only, then the sign has its more normal meaning. N = 5 means that N takes the value of 5. IF N = 6 means that the test will succeed when N is exactly 6, and so on.

Fortified with this new knowledge, we can now take a look at a more elegant program for finding a total. This one has a title, and it displays for each entry an item number, and shows the total along with the number of items. We're starting now to get away from the type of program that a calculator can carry out, and getting into the region of work that truly belongs to a computer. The program is shown in Fig. 3.6.

In line 2∅, variables N (the item number) and TL (the total) are set to zero. We've used TL for total and *not* TO because TO is a *reserved word* – one of the words that the computer recognises as an instruction. If we use an instruction word as a variable name, then the computer always takes it as an instruction – it can't be expected to know what you intended. The lines 3∅ to 7∅ then print a title and instructions, and the real action starts in line 8∅. Line 8∅ is N = N + 1, so that the value of N, the item number, which started as ∅ in line 2∅, becomes 1. This way, line 9∅ will print ITEM 1, so that you can then input this item. Assuming that the first item is not zero, then line 11∅ will make the total equal to this first number, and line 12∅ will print the total to date and the number of items.

Line 13∅ is a bit of a novelty, though. Rather than testing for J = ∅, we test for J *not equal to* zero. This allows us to use only one

```
 5 REM MORE TOTAL
 10 CLS:PRINT
 20 N=0:TL=0
 30 PRINTTAB(17)"TOTALS"
 40 PRINT:PRINT
 50 PRINT"THE PROGRAM WILL DISPLAY A TOT
AL FOR

 60 PRINT"ALL THE NUMBERS YOU TYPE. TYPI
NG 0"
 70 PRINT"WILL END THE PROGRAM."
 80 N=N+1
 90 PRINT"ITEM ";N;"  ";
 100 INPUT J
 110 TL=TL + J
 120 PRINT"TOTAL IS ";TL;" FOR ";N;" ITE
MS
 130 IF J<>0 THEN 80
 140 END
```

Fig. 3.6. An improved running-total program.

GOTO, because we GOTO8∅ only if a zero has *not* been entered. At line 8∅, the value of N, which was 1, is knocked up to 2, and lines 9∅ to 12∅ are repeated. Once more, line 13∅ tests the value of J, repeating the steps of the loop if a ∅ has not been entered. If a zero has been entered (it will count as an item), then the program ends in line 14∅.

Suppose you don't want to count the zero as an item? One way out is to have a line:

1∅5 IF J = ∅ THEN 14∅

and then make line 13∅ read:

13∅ GOTO 8∅

That's why we use line numbers that step in tens. If you have second thoughts about the way that a program carries out its actions, you can put in a new line between some previous ones. Later on, though, we'll turn our attention to planning a program so that this sort of thing should never be necessary – well, hardly ever. Chapter 6 attends to that.

Next item, please

Loops are such an important part of computer programming that a special loop instruction is provided. The keywords for this type of loop are FOR...NEXT, and we'll see how it works most easily by

```
5 REM FOR NEXT
10 CLS:PRINT:TL=0
20 FOR N=1 TO 10
30 PRINT"ITEM ";N;" IS ";
40 INPUT J
50 TL=TL+J
60 PRINT"TOTAL IS NOW ";TL
70 NEXT
80 PRINT"N IS NOW ";N
90 PRINT"FINISHED"
100 END
```

Fig. 3.7. Using the FOR...NEXT loop.

looking at an example. Fig. 3.7 is a short version of a totalling program which will deal with only ten items. Line 2∅ contains the instruction FOR N = 1 TO 1∅ which means that a count will be organised for you. A variable N (as always, we can pick our own name for it) will start at 1 and count as far as 1∅ in steps of 1. Each new count will start when the word NEXT is met in the program, so that every instruction that we type on lines between the FOR in line 2∅ and the NEXT in line 7∅ will be repeated ten times. When the tenth item has been dealt with, the effect of NEXT is to go back to line 2∅ with N equal to 11. Line 2∅ will then reject this value because it is more than the limit of 1∅ that has been set. That's why line 8∅, which is the next line after the loop, prints the value of N as being 11 as the program ends.

You don't have to decide at the time when you write the program what the maximum value of N has to be, because you can make this value a variable as well! Take a look at Fig. 3.8, in which the maximum value is a variable MX which is fed in by the user. This is a

```
5 REM VARIABLE MX
10 CLS:PRINT:TL=0
20 INPUT"NUMBER OF ITEMS-";MX
30 FOR N=1 TO MX
40 PRINT"ITEM ";N;" IS ";
50 INPUT J
60 TL=TL+J
70 NEXT
80 PRINT"AVERAGE VALUE IS ";TL/MX
90 END
```

Fig. 3.8. A FOR...NEXT loop which uses a variable to decide the maximum number.

short (no explanations) average number finder for as many values as you like to use. In line 2∅, you are asked for the number of items, and when you type the number and press RETURN, the number is stored as MX. Don't use a large number when you are testing this one, or you'll be there all night! The loop then starts in line 3∅, and you can input each amount that you want to put in for the average. Line 6∅ totals the numbers, and keeps doing so, without displaying the total, until the loop ends. This will happen when you have entered the last of the items that you specified in line 2∅. Line 8∅ then prints the average value. This is the ordinary arithmetical average, equal to the total of the items divided by the number of the items.

You can, incidentally, still check the value of variables after the program has finished. Provided that you don't start to RUN again, or get rid of the program by using NEW, you can type, for example:

PRINT TL

and then press RETURN to get the value of the total TL. It's a point that is well worth remembering if you want to check what a program has got up to.

We seem to have spent a lot of time and space on loops that use numbers. Let's show some of the possibilities that exist when we use strings. To do so, we need to introduce a new instruction, READ. READ always has to be used along with another instruction word, DATA, and the two need not be placed anywhere near each other in the program. Fig. 3.9 shows how you could print a shopping reminder list.

```
5 REM READING
10 CLS:PRINT
20 FOR N=1 TO 10
30 READ D$
40 PRINT D$
50 NEXT
60 DATASALT,MUSTARD,VINEGAR,PEPPER
70 DATATURMERIC,CHILI,CUMIN,FENNEL
80 DATAROSEMARY,CINNAMON
```

Fig. 3.9. Using READ...DATA to make a list of items.

The loop starts in line 2∅, and specifies ten items. The items themselves are arranged in lines 6∅ and 7∅, in order, with the word DATA before the start of each set of items. Each item is separated from others by a comma, but you don't need to put a comma between the word DATA and the first item in each line of the list. You don't need quotes around these string items, either.

In line 3∅, the instruction READ D$ means that the computer must look for the DATA lines and read an item from one of them. This reading will start with the first item in the lowest numbered DATA line. The next READ will be of the next DATA item in order, and so on, going from item to item and line to line. By putting

READ D$ and PRINT D$ in the loop together, we read all ten items of the DATA and print them all.

What happens if we don't have ten items? Try it. Type 7∅ and press RETURN. This will remove line 7∅, so that only five of the items remain. When you RUN the program now, you will get a message OUT OF DATA after the five items have been printed. This message means that you tried to read more items than were put into the DATA lines, and the computer could not continue. If you had too many items, however, there would be no error message – but it would read only ten of them!

A modification to the FOR...NEXT instruction allows you to perform the count in steps other than 1. By typing STEP followed by a number (negative, positive or fractional), you can specify whatever step you like. For example:

```
FOR N = 1∅ TO 1 STEP -1
```

will cause a count down from 1∅ to 1, and:

```
FOR N = 1 TO 1∅1 STEP 2
```

will count 1,3,5...to 1∅1. We'll illustrate this use of STEP, which must always follow the FOR conditions, in later programs. If you simply want a STEP of 1, then you can omit STEP.

Moving to another point, by typing a new line:

```
45 RESTORE
```

you can change the program rather noticeably. It looks rather repetitive now, doesn't it, when you run it? RESTORE has the effect of resetting the list back to its start, so that you are reading the first item each time. This instruction is useful if we have to read a list more than once in the course of a program.

Suppose you wanted to be able to keep adding or deleting items from the DATA list? The program of Fig. 3.9 isn't so useful then, because you have to alter the number in the loop at line 2∅ each time. We can get around this problem by using a *terminator*. A terminator is an item which can be recognised by an IF test and which can then be used to stop the program. It must be an item which would never normally belong to the list, and we can use items like X or XX for string lists, and ∅ (sometimes 999 or −1) for number lists.

Try the program in Fig. 3.10. There's no FOR...NEXT loop used here, because the program can be used to read and print as many items as there are in the DATA list, and this need not be a set

```
5 REM UNLIMITED!
10 CLS:PRINT
20 READ D$
30 IF D$="X"THEN 60
40 PRINT D$
50 GOTO 20
60 END
70 DATACHABLIS,SAUTERNE,BURGUNDY
80 DATABEAUJOLAIS,MUSCATEL
90 DATALIEBFRAUMILCH,PIESPORTER
100 DATA X
```

Fig. 3.10. An alternative method of using READ...DATA with a 'terminator' item.

number. Each item is read, and in line 3∅ is checked to find if it is the terminator item. If it is, the program ends in line 6∅ but, if not, the program loops back to line 2∅ so as to read another item from the list. This action can be done more efficiently by a REPEAT... UNTIL loop instruction, but my early ORIC did not have this built in.

Number handling

Computing isn't all about numbers, as most of this book will show, but the ability to carry out actions with numbers is useful. This is no place to teach mathematics, however, so it you find that you are getting into completely unfamiliar territory, then it's better to move on.

To start with, we need to look at the two different types of number variables. We have used variables like N,JJ or R2, consisting of a letter by itself or followed by another letter or number. Variable names like this represent *real numbers*, meaning numbers that can contain fractions, and which can take a very large range of possible values. Examples are 26.35,-212.7, ∅.292 or 3.6E7 (3.6×10^7 as we usually write it). There is another class of number variable available on your ORIC, the *integer*. An integer variable uses the same kind of letter-number codes as we can use for a real number, but followed by the % sign. An integer is a whole number which can take a limited range of values between −32768 and +32767, all whole numbers.

What's the point? It's all tied up with the way that computers store numbers. An integer can be stored using only two units of memory, and the value of an integer variable is *always exact*. Any arithmetic that you carry out using integers is precise, and any programs that use integers run much faster and use less memory than programs that use real numbers. The ORIC has the restriction, however, that integers cannot be used as the counter numbers in FOR...NEXT loops. This is unfortunate, because this is one of the examples in which the use of integers greatly speeds up the action of the computer.

As an example, take a look at Fig. 3.11. Line 4ø calculates .2/2,

```
 10 CLS:PRINT
 20 A=4.6:B=4.7
 30 C=.2:D=2
 40 IF C/D=B-A THEN PRINT"CORRECT":GOTO6
0
 50 PRINTC/D;" IS NOT EQUAL TO ";B-A
 60 Y%=5:Z%=3:X%=10
 70 IFX%/Y%=Y%-Z%THEN PRINT"INTEGERS ARE
 CORRECT":GOTO90
 80 PRINTX%/Y%;" IS NOT EQUAL TO ";Y%-Z%
 90 END
```

Fig. 3.11. Numerical accuracy – if you thought that computers were for mathematical wizards, think again. Note the effect of using integers.

which should be equal to .1, and B − A, which should also be .1, and compares the results. They ought to be equal, but the computer finds that they are not, and prints line 5ø! The arithmetic in line 7ø is correct, however. Why should there be a difference? The reason is due to the way that the computer stores numbers. An integer is stored in an exact way, because it uses no fractions. All real numbers are stored in two parts, a fraction and a multiplier. This is like the way that large numbers are entered or displayed in a calculator, keying in 1.6EXP6 (or 1.6E6) instead of 1600000, for example. The trouble with this method as it is applied to computers is that the computer does not use a decimal counting scale, and every fraction, unless it's one that is a power of 2 like 1/2, 1/4, 1/8, 1/16 and so on, is stored as an approximate value and rounded up or down when it is displayed on the screen. It's rather like the problem that we have when we try to express fractions like 1/3 as decimals – the result is never exact no matter how many places of decimals we use.

Now the approximation may not be serious. If the computer stores .1 as .100000001 or as .099999999, and the screen shows .1 for each of these numbers, we are none the wiser, because the number on the screen looks exact enough. When we compare numbers using =, however, the computer expects them to be identical, and it is the stored quantities that are compared. When we use integers, we don't have to worry about these problems.

Integers aren't always convenient, though. It would be awkward to deal with money sums in exact pounds, for example, and even if we converted to pennies, sums between −£327.68 and +£327.67 would be rather limiting. The problems arise only when sums are compared, so what we have to do is to limit or round off the numbers that we compare. For financial work, we would round off to the nearest penny. If, for example, we clip each number to two places of decimals before carrying out an equality test, this will catch some types of errors. Try the example in Fig. 3.12. Before the quantities are compared, they are rounded up, multiplied by ten, and the instruction INT is used.

```
 5 REM CUTOFF
 10 CLS:PRINT
 20 A=4.6:B=4.7
 25 K=B-A+.0001
 30 C=.2:D=2
 40 IF INT(10*C/D)=INT(10*K)THEN PRINT"C
ORRECT":GOTO60
 50 PRINTINT(10*C/D);" DOES NOT EQUAL ";
INT(10*K)
 60 END
```

Fig. 3.12. Comparing numbers *after* rounding.

INT is a *number function* – an action that operates on a number and produces another number. What INT does is to remove fractions. INT(1.6) is 1, INT(12.75) is 12, and INT(−23.6) is −23. Now if we have the number .1, then 1∅*.1 = 1, and INT(1) = 1. If the computer stores the number as 1.00000001, the INT removes the fraction which causes all the trouble when numbers are compared. If, as generally happens with the ORIC, the number is stored as .99999999, then adding .0001, .001 or anything of this sort (preferably small) will bring the number up to a value which INT will round off. Always use rounding of this type in money sums – multiply by 100 to get the amount into pennies, round up, use INT and then if you want to compare, do so. You can then divide by 100 again to obtain a usable amount.

For solving equations, you may be able to make use of some other number functions, as shown in Fig. 3.13. Fig. 3.14 shows examples of how formulae can be 'translated' into BASIC terms. The important point here is the order in which actions will be carried out. Deciding the sign of a number has first priority, followed by raising to a power, then multiplication and division, and finally addition and subtraction. If all actions have equal priority, then a left-to-right order is used. We can enforce a different order by enclosing between

Number Function	*Meaning*
ABS	Gives absolute value, meaning that the sign is removed. ABS(−6) = 6.
ATN(X)	Gives the angle, measured in radians, whose tangent value is X. Y = ATN(X)
COS(X)	Gives the cosine of angle X. X is expressed in radians. Y = COS(X)
EXP(X)	Gives the size of the quantity e^x. Y = EXP(X)
LOG(X)	Gives natural (base e) logarithm of X. Y = LOG(X)
SGN(X)	Finds the sign of X. Gives 1 if X is positive, −1 if X is negative, and ∅ if X is zero. Y = SGN(X)
SIN(X)	Gives the sine of the angle X. X is expressed in radians. Y = SIN(X)
SQR(X)	Gives the square root of X. Y = SQR(X)
TAN(X)	Gives the tangent of angle X. X must be expressed in radians. Y = TAN(X)

Conversions:
To convert degrees to radians, divide the angle in degrees by 57.296.
To convert natural logarithms into ordinary (base 1∅) logs, divide by 2.303.

Fig. 3.13. The other number functions of the ORIC, with brief examples. If you don't understand them, feel free not to use them!

Formula	*BASIC version*
$y = mx + c$	Y = M*X + C
$y = a^2 + b^2$	Y = A↑2 + B↑2
$y = \sqrt{a^2 + b^2}$	Y = SQR(A↑2 + B↑2)
$X = a\cos B + c\cos D$	X= A*COS(B) + C*COS(D)
$a = \log_{10} b$	A = LOG(B)/2.303

Fig. 3.14. Translating formulae into BASIC – some examples.

brackets anything that we want to be done first. We can even use brackets within brackets (nested brackets).

For example, an expression like:

5+Y−X↑2*3

will be evaluated as follows. The X↑2 (value of X squared) is found first, and then this quantity is multiplied by 3. This is then subtracted from 5 plus (value of Y). Now if we had used brackets in the expression like this:

(5 + Y − X)↑2*3

then the size of 5+Y−X would have been worked out, squared, and the result of this multiplied by three.

A number function that was not included in the list of Fig. 3.13 was RND. RND means 'random' and the keyword RND is used to generate a number at random. If we use an instruction such as:

PRINT RND(1)

then what is printed is a fraction between ∅ and 1, but *always* less than 1. This is a 'random fraction', meaning that you can't predict what number will be printed – though it isn't truly random in a sense that a mathematician would expect. We can use RND(1) to generate random numbers in any range we like. Suppose we want a whole number picked at random between 1 and 1∅. By using RND(1), we will get a number that can range from almost ∅ to almost 1, and multiplying this by 1∅ will give a number which will be somewhere between (almost) zero and (almost) 1∅. Taking the INT of this number will produce something which can range from ∅ to 9, and adding 1 will then give a number that can be anything from 1 to 1∅. Fig. 3.15 shows a program that illustrates the numbers that can be generated by this method. Run this several times, and you should see that numbers are produced more or less at random (though my ORIC seemed rather fond of 9!).

```
10 REM RANDOM
20 X=RND(1)
30 PRINT"RANDOM NUMBER IS ";X
40 Y=10*X
50 PRINT"TEN TIMES THIS IS ";Y
60 Z=INT(Y)
70 PRINT"THIS ROUNDS TO ";Z
80 PRINT"ADD 1 TO GET ";Z+1
90 PRINT"NOW TRY ANOTHER ONE"
100 END
```

Fig. 3.15. Generating a random number whose value lies between 1 and 1∅.

Do-it-yourself functions

If the list of functions in Fig. 3.13 is not enough for you, the ORIC allows you to make up your own. These functions that you can make up for yourself are called 'user-defined functions', or just *defined functions*, and you have to type what you want them to be, starting with DEF FN. This signals to the computer that what follows is a definition of a new function that it will have to memorise and use.

Following DEF FN, you need a name for the function. This has to be a single letter (it won't be confused with a variable that uses the same letter) – anything from A to Z. Following that, within brackets, must be a variable name. This second variable name is used to describe the function; it is the letter that will be used in the equation that shows what the function does. After all this, we need an equals sign, and then the function itself.

An example makes it look a lot simpler, so take a look at Fig. 3.16. In line 1Ø we have:

DEF FNA(K) = K↑2

```
5 REM DEFINED FUNCTION 1
10 DEF FNA(K)=K^2
20 PRINT FNA(5)
30 X=FNA(5)+FNA(12)
40 PRINT SQR(X)
```

Fig. 3.16. A simple defined function. This one works out the squares of numbers.

This means that we are defining a function called A whose action will be to square a variable K. If K is 3, for example, the action of the function will be to produce 3^2, which is 9. The command sign for raising to a power, remember, is the up-arrow. Now the useful feature of a defined function is that we don't actually have to use K at all! Anything that we put between the brackets when we make use of the function, providing that it is a number or a number variable, will be treated the way that K was treated in the definition. If we ask the computer to work on the number 5, for example, we do this (line 2Ø) by the instruction PRINT FNA(5). This means – work out what

```
10 DEF FNQ(X)=X^2+3*X+2
20 PRINT"ENTER A NUMBER";
30 INPUT Z
40 PRINTZ;"^2+3*";Z;"+2 IS ";FNQ(Z)
50 END
```

Fig. 3.17. A more useful defined function that works out the value of an equation.

DEF FNA(R) = 3.14*R↑2	Area of circle
DEF FNB(R) = 4*3.14*R↑2	Area of sphere
DEF FNC(R) = (4*3.14*R↑3)/3	Volume of sphere
DEF FND(F) = 3E8/F	Finds wavelength for frequency F
DEF FNE(C) = 1/(2*3.14*F*C)	Reactance of capacitor C – must supply F
DEF FNF(L) = 2*3.14*L	Reactance of inductor L – supply F
DEF FNG(F) = 1/(2*F*SQR(L*C))	Frequency of resonance – supply L, C
DEF FNH(X) = (-B + SQR(B↑2 - 4*A*C))/(2*A) }	Quadratic roots – supply A, B, C
DEF FNJ(X) = (-B - SQR(B↑2 - 4*A*C))/(2*A) }	(X is dummy variable, not used)
DEF FNK(P) = (R1*R2)/(R1 + R2)	2 resistors in parallel
DEF FNL(P) = 1/(1/R1 + 1/R2 + 1/R3)	3 resistors in parallel
DEF FNM(D) = 8.84E - 12*A/D	Capacitance of parallel plates – A supplied
DEF FNN(X) = (INT(X*100))/100	Rounds off to two places
DEF FNP(N) = (INT(X*10*N))/10*N	Rounds off to N places

Fig. 3.18. Table of some defined functions.

function A does with 5 in place of K. Since the function squares, this gives 5^2, which is 25. Lines 3Ø and 4Ø show a more elaborate use of this same function.

We don't, of course, need to use a defined function to square a number, because we have the up-arrow command to do just that. Fig. 3.17 shows a defined function that is more useful. The function is an equation, $X^2 + 3X + 2$ and we can put any number we like in place of X. The computer will then work out the value of the amount and print it. All we need to use is FNQ(Z) where Z is the number.

The table in Fig. 3.18 shows some defined functions that you may find useful. Note that only number or number variables can be used, and that only one value can be 'passed' to the equations – the value that is enclosed by the brackets. We can't for example, have FN(A,B,C), but we *can* have DEF FNX(A) = SQR(A↑2 + B↑2) providing that B is a variable that has a value allocated to it in the program. We can use any variable name in place of A, however.

Chapter Four
String Handling

From what we have done already, you will have seen a difference between the way we use numbers and strings. A string is anything that is placed between quotes in a program line or entered as a string variable in an INPUT or READ instruction. The differences go deeper than this, though, because we can have N$ = "2" and N = 2. Printing these makes them appear identical, but we can't perform the action of multiplication (just to take an example) on N$, only on N. Any attempt to PRINT 2*N$ will give the error message TYPE MISMATCH ERROR, though PRINT 2*N gives 4 as we might expect. This is because number and string values are stored in the computer in entirely different ways.

A computer is a machine which can deal with numbers only. Anything that we do not treat as a number – meaning that we do not perform arithmetic on it – is put into the form of a number code, using a number to represent each character. The code that is used by most computers is called ASCII (pronounced ASKEY) code. The name comes from the initial letters of American Standard Code for Information Interchange. This code uses a different number to represent each character on a typewriter or computer keyboard, and the codes that are used by the ORIC are shown in Fig. 4.1. As you type quantities like ORIC or 27.6 into a program line, they are stored temporarily as a set of ASCII code numbers. When you press RETURN, however, quantities that are numbers or are assigned to number variables are converted into number form of fraction and multiplier, but strings are not. In this way, ORIC is stored as the codes 79,82,73,67, but 27.6 is stored as five units coded in a very different way. The important point is that every real number is stored as five sections, every integer as two, no matter what the numbers happen to be. The way in which numbers are stored allows the computer to carry out arithmetic. The way in which strings are stored, using ASCII codes, allows the computer to carry out a different set of actions that we call *string functions.*

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	–	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	¦
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	

Fig. 4.1. The ASCII codes for the characters that ORIC uses. Some numbers correspond to characters that appear as one character on the printer, another on the screen, and another on the keyboard!

LEN and VAL

LEN and VAL are two instruction words that relate to strings only. LEN means 'find length', and the length of a string is the number of characters in the string, equal to the number of ASCII codes used to store the string. Fig. 4.2 illustrates this in action – whatever you type,

```
5 REM LEN
10 CLS:PRINT
20 PRINT"PLEASE TYPE A SHORT TITLE";
25 INPUT A$
30 L=LEN(A$)
40 PRINT"THERE ARE ";L;" CHARACTERS"
50 PRINT "IN ";A$
60 END
```

Fig. 4.2. Using LEN to find the number of characters in a string.

numbers, letters, punctuation marks, in line 2Ø is assigned to the string variable name A$. In line 3Ø, the number variable L is assigned to LEN(A$), which means that the value L is equal to the number of characters in A$. The amount is printed in line 4ø. We can make use of LEN(A$) as we would use any other number, so that we say this is a string function that returns a number.

VAL is another string function which returns a number, but in this case it's the number represented by the string. If you read a number into a string variable (by using READ A$ with DATA 99, for example) or input a number to a string variable (INPUT A$), then the number is stored like any other string, in ASCII code form. This prevents us from using it like a number, though when we print it on the screen it looks like a number. VAL converts such strings back into number variable form, as the example in Fig. 4.3 shows. When VAL is used, the result is stored as a number and can be used as a

```
5 REM VAL
10 CLS:PRINT
20 N$="21.67"
30 PRINT N$;" TIMES 2 IS ";2*VAL(N$)
40 PRINT"WE CAN'T USE ";N$;" DIRECTLY"
50 PRINT N$*2 :REM CAUSES ERROR
60 END
```

Fig. 4.3. How VAL extracts a number value from string form. STR$ does the opposite (see Chapter 10).

number. We can, for example, have V = VAL(N$) and then use the number variable V.

VAL will operate only if the string contains a number in ASCII code form and if that number is not 'embedded'. As Fig. 4.4 shows,

```
5 REM VAL EMBEDDED
10 CLS:PRINT
20 A$="12 ACACIA AVENUE"
30 B$="ROOM 504"
40 PRINT VAL(A$)
50 PRINT VAL(B$)
60 PRINT"VALUES AS ABOVE"
70 END
```

Fig. 4.4. Using VAL with strings that contain numbers and letters.

VAL will convert into number form a set of figures which occur at the start of a string, before any letter. It will not, however, be able to convert figures which follow letters, as the example also shows.

There is one arithmetic action which will not cause an error report when you carry it out on a string, the 'addition'. The action on strings is not the same as the action on numbers, however. Fig. 4.5 shows the differences when both strings are numbers in string form. Using the + sign between the two strings causes the strings to be joined, so that "12" + "34" is just "1234". If we extract number

```
5 REM JOINING
10 CLS:PRINT
20 A$="12"
30 B$="34"
40 PRINT A$+B$;" JOINED"
50 PRINT VAL(A$)+VAL(B$);" ADDED"
60 END
```

Fig. 4.5. Joining or *concatenating* strings.

values by using VAL, then the arithmetical result, 46 is what we get. The string-joining operation is called *concatenation*, and it can be useful to put strings into order with punctuation marks, as Fig. 4.6 shows.

```
5 REM CONCATENATION
10 INPUT"SURNAME, PLEASE";SN$
20 INPUT"FORENAME, PLEASE";FR$
30 NM$=SN$+", "+FR$
40 CLS:PRINT:PRINT NM$
50 END
```

Fig. 4.6. A typical use for concatenation. Other examples are illustrated in Chapter 7.

ASC and CHR$

ASC is a shortened version of ASCII, and the ASC of a string is a number, its ASCII code. If the string consists of more than one letter, then only the code for the first letter will be printed. Fig. 4.7

```
 5 REM ASC
 10 INPUT"PRESS A KEY, PLEASE(THEN RETUR
N)";K$
 20 PRINT"ITS CODE IS ";ASC(K$)
 30 GOTO10
```

Fig. 4.7. Using ASC to find the code for the first character of a string.

shows ASC in use, in a program that will print the ASCII code for any key that you care to press. If you type a word rather than just a key, then the code is the ASCII code for the first letter. Note, incidentally, the codes that are generated by keys such as RETURN, ESC, etc.

The use of ASCII code explains how the greater-than and less-than signs can be used along with strings. The ASCII code numbers for upper-case (capital) letters start at 65 for A and go up to 90 for Z. The lower-case (small) letters use codes that are 32 greater, so that a is 97 and z is 122. The numerical order of ASCII codes for letters is

```
  5 REM ORDER
  10 INPUT"TYPE A LETTER...";A$
  20 INPUT"AND ANY OTHER...";B$
  30 IFA$>B$THEN C$=A$:A$=B$:B$=C$
  40 IF A$=B$THEN PRINT"THEY ARE IDENTICA
L":END
  50 PRINT A$;" COMES BEFORE ";B$
  60 END
```

Fig. 4.8. Putting strings into alphabetical order. This is also the order of the ASCII codes.

therefore capitals first and alphabetical order. Fig. 4.8 shows how letters can be put into alphabetical order. Two letters are entered and assigned to A$ and B$. The comparison is done in line 3∅. If A$ has a greater ASCII code number than B$, then it is out of order, and the strings are exchanged. This is done by making the string C$ hold the value of A$, assigning A$ to the value of B$, and then assigning B$ to the value of C$ to finish the swap. Line 4∅ prints an explanation if the strings are equal, otherwise line 5∅ shows the correct order.

There are two interesting points about this program. One is that it works with words as well as with letters. If you make A$ = "ART" and B$ = "ARM", then these words, which use the same starting letter, are still sorted into their correct order. The use of the signs < and > (or =) will cause the ASCII codes of the strings to be compared, character by character, until a difference is found or until the two are proved identical.

CHR$ is a string function whose action is just the opposite of ASC. The result of CHR$, which has to be followed by a number in brackets, is a letter. It's the letter whose ASCII code is the number, so that CHR$(68) is D. This character, which is a string, remember, can be printed or assigned to a string variable as we please. Fig. 4.9

```
5 REM CHR$
10 FOR N=32 TO 98
20 PRINT CHR$(N);"  ";
30 NEXT
40 END
```

Fig. 4.9. Using CHR$ to find the character corresponding to a number.

shows how an alphabet can be printed out without mentioning letters in the program. We can use this also to print words, and it's useful as a way of reducing cheating in games programs. Fig. 4.10 shows a message written as ASCII codes on a DATA line. Unless you know the ASCII codes by heart, it's not obvious what it will print when you run it. Try it!

```
5 REM CODES
10 FOR N=1TO20
20 READJ:PRINT CHR$(J);
30 NEXT
40 END
50 DATA79,82,73,67,32,87,73,78,83,32
60 DATA69,86,69,82,89,32,84,73,77,69
```

Fig. 4.10. Using ASCII codes to conceal a message that is printed using CHR$.

Left, right and middle

A particularly important string action is called *string slicing*. Because a string is stored in the memory as a set of ASCII codes, we can take any part of a string that we want by counting the number of characters. A part of a string chosen like this is called a *slice*, and the ORIC has three slicing actions which use the keywords LEFT$, RIGHT$ and MID$.

LEFT$ has the effect of producing bits of the string starting at the left-hand side. We can specify how many characters we want to take, using a number, and we can print the slice or assign it to another string variable. For example, if X$ = "ORIC-1", then LEFT$(X$,1) is "O", and LEFT$(X$,2) is "OR". Fig. 4.11 is an example of slicing,

```
5 REM LEFT
10 CLS:PRINT
20 INPUT"TYPE YOUR NAME,PLEASE";NM$
30 FOR N=1TO LEN(NM$)
40 PRINT LEFT$(NM$,N)
50 NEXT
60 END
```

Fig. 4.11. Taking the left slice of a string.

using an increasing number between slices. By using LEN(NM$) in line 3∅, we slice no more than the total length of the string. If you attempt to slice more characters than is possible, such as by using PRINT LEFT$(X$,50) then you simply get the whole of X$ printed, assuming that it has fewer than 5∅ characters.

LEFT$ will slice a number of characters starting with character 1 at the left-hand side, and RIGHT$ will act similarly, but with the count starting at the right-hand side. Fig. 4.12 shows this one in action. Whatever you type in answer to line 2∅ is assigned to NM$, and then a loop starts in line 3∅, taking values of N which range from 1 to the total number of letters in NM$. Line 4∅ prints the sliced piece of string, using TAB numbers that ensure that the string ends at the right-hand side of the screen. If you find this one a bit hard to

```
5 REM RIGHT
10 CLS:PRINT
20 INPUT"TYPE YOUR NAME, PLEASE";NM$
30 FOR N=1TO LEN(NM$)
40 PRINT TAB(39-N)RIGHT$(NM$,N)
50 NEXT
60 END
```

Fig. 4.12. A right slice action.

swallow, think of the effect of each different value of N. If N is 2, for example, then printing starts at TAB(37), and is of the last two letters of the word. Since the screen columns are numbered ∅ to 39 across, starting from the left-hand side, this places the last letter at the right-hand side of the screen.

The real workhorse of the slicing instructions, however, is MID$. This allows us to slice as many letters as we please, starting where we want. The syntax is that the sliced string is equal to MID$ (string, start, number), where string means the string you want to slice; start is the number of the first character, numbering from the left of the string (the first character is numbered 1); and number is the number of characters you want to select, reading from left to right. For example, if X$ = "ORIC-1", then MID$(X$,2,3) is "RIC". The slice starts at character 2, which is R, and is of three characters, R, I and C.

Like the other two slicing instructions, MID$ allows us to use number variable names or expressions in place of numbers. This allows us to work out more interesting tricks with letters as Fig. 4.13

```
5 REM MID
10 CLS:PRINT
20 INPUT"TYPE YOUR NAME, PLEASE";NM$
30 MN$="":L=LEN(NM$)
40 FOR N=L TO 1 STEP -1
50 MN$=MN$+MID$(NM$,N,1)
60 NEXT
70 PRINT"THAT BECOMES ";MN$;"!"
80 END
```

Fig. 4.13. Using MID$ to reverse the order of letters in a name.

shows. This one performs the trick of reversing the letters of a name. Line 3∅ sets up ready by making a new empty string, MN$, and finding L, the number of characters in the name. The loop starting in line 4∅ then picks letters out of the name NM$ and puts them one by one into the new string MN$. Because the loop starts with L and steps downwards, however, the first letter that is selected by MID$(NM$,N,1) is the last letter of NM$. By using 1 in the MID$

instruction, we ensure that only one letter is picked at a time. This is put into MN$ and each time the loop goes round the value of N is made one less and the letter corresponding to that value is added to the end of MN$. The result is then printed in line 7∅.

The way in which MID$ can be used along with variable names and expressions makes it a very useful instruction for coding and selecting. Just to take an example, suppose you wanted to write a program so as to teach someone to recognise the number (1 to 26) for each letter of the alphabet. You could do this in the form of a game, so that the computer selected a number at random, in the correct range, and then asked, for example:

WHICH LETTER IS NUMBER 16?

The user would then guess, and the guess would be compared with the correct answer.

Now you might approach such a program by having a string variable name for each letter – but where do you go from there? The easy method is to define a string AL$ which consists of all the letters of the alphabet. To find letter 12 in this string, then, all you have to do is to use MID$(AL$,12,1). The suggested program in Fig. 4.14,

```
 5 REM ALPHA
 10 CLS:PRINT:AL$="ABCDEFGHIJKLMNOPQRSTU
VWXYZ"
 20 PRINT"THIS PROGRAM IS INTENDED TO TE
ST YOUR"
 30 PRINT"KNOWLEDGE OF THE POSITION OF L
ETTERS"
 40 PRINT"IN THE ALPHABET. A NUMBER WILL
 APPEAR"
 50 PRINT"AND YOU ARE INVITED TO TYPE TH
E "
 60 PRINT"LETTER TO WHICH IT CORRESPONDS
,"
 70 PRINT"TAKING A=1,B=2 ETC.":PRINT:PRI
NT"READY NOW...."
 80 NR=1+INT(25*RND(1))
 90 S$=MID$(AL$,NR,1)
 100 PRINT"NUMBER IS ";NR
 110 INPUT"LETTER IS ";Y$
 120 IF Y$=S$ THEN PRINT"YES...TRY ANOTH
ER":GOTO 80
 130 PRINT"NO..TRY ANOTHER":GOTO80
 140 END
```

Fig. 4.14. A guessing game based on the use of MID$ with the alphabet.

defines AL$ as all 26 letters of the alphabet in order, and after a set of instructions in lines 2∅ to 7∅, a random number in the range 1 to 26 is picked in line 8∅. This is then used in line 9∅ to pick out the letter corresponding to the number, and the number is then printed (line 1∅∅). You are then invited to type the letter corresponding to the number that you see on the screen in line 11∅. If your answer corresponds to the MID$ slice, you are informed of this in line 12∅, and another number is produced. If you do not score, then line 13∅ is executed.

There are two obvious improvements that you could make to this. One is that no score is kept. What we need is a number variable that starts at zero and is incremented each time line 12∅ is executed. I'll leave you to attend to that and perhaps to print the score each time round. You might also want to print the number of tries – a number variable which also starts at zero and which is increased each time there is an answer.

The other important point concerns the way in which the letter is put into the machine. We have used INPUT, which requires you to type the letter and then press the RETURN key. This gives you time for second thoughts, because what you have typed can be deleted and replaced until you press RETURN.

GET it?

As it happens, there is a more immediate way of typing one single letter or number with the ORIC. This uses the instruction word GET, which must be followed by a variable name. When the computer encounters GET, it stops computing and scans the keyboard, trying to detect if any key is pressed. If no key is pressed, it will keep scanning the keyboard, waiting for you. When a key, any key, is pressed, the program then continues. The 'value' of the key is assigned to the variable name that was used after the GET instruction. The SHIFT and CTRL keys have no effect, but all of the other keys will have some effect. Try the example in Fig. 4.15 to see

```
5 REM GET 1
10 CLS:PRINT
20 PRINT"PRESS ANY KEY"
30 GET A$
40 PRINT"YOU PRESSED ";A$
50 END
```

Fig. 4.15. How GET is used with a string variable. GET will always cause the computer to wait for you.

this in action. Note that the ESC key will operate the GET action, but some of the keys, including ESC, will not produce anything on the screen. This is because their ASCII codes do not correspond to characters that can be printed. The RETURN and the cursor (arrowed) keys, for example, will not print anything, nor will the spacebar.

We can also use GET with a number variable, but when we do this only a number key will be accepted. Try the example in Fig. 4.16,

```
5 REM GET 2
10 CLS:PRINT
20 PRINT"PRESS A NUMBER KEY.."
30 GET A
40 PRINT"IT WAS ";A
50 END
```

Fig. 4.16. Using GET with a number variable. The computer will hang up if you type a letter.

first with a number key and then with a letter. If you press a letter key, you will get the SYNTAX ERROR message, which will always stop a program. To avoid this we generally use GET A$, and if a number is needed we use V = VAL(A$) to extract the value of the number. By programming in this way, the program will not be halted if the wrong key is pressed. Fig. 4.17 illustrates how this is done.

```
 5 REM GET 3
 10 CLS:PRINT
 20 PRINT"PRESS ANY KEY...."
 30 GET A$
 40 IF VAL(A$)=0 THEN PRINT"THE CHARACTE
R IS ";A$:GOTO60
 50 PRINT"THE NUMBER IS ";A$
 60 END
```

Fig. 4.17. Using GET A$ along with VAL so that any key can be accepted.

These programs also illustrate one of the most common uses for GET. Every now and again, you want a program to wait for you. Perhaps you want to leave several lines of instructions on the screen so that you have time to read them, or you want to leave time so that you can prepare a cassette. Using GET allows you to do this, because you can then make the last instruction line read:

"PRESS ANY KEY TO PROCEED"

and then use GET A$. The program will be suspended, waiting for you, and will continue only when you press a key. We'll look at the use of GET in menu choices later on, in Chapter 5.

Pause a while

A useful feature of computing is a pause, as we have seen. Sometimes, however, we want a timed pause rather than an indefinite pause of the type that uses GET. We can create such a pause in a program simply by using a loop like FOR N = 1 TO 1ØØØ:NEXT, but the ORIC allows you to use a more direct instruction for this purpose. This is WAIT, and it has to be followed by a number, not in brackets. The number is in hundredths of a second, so that WAIT1ØØ produces a pause of one second, WAIT1ØØØ produces 10 seconds, and so on.

Control codes

We have seen how the numbers 32 to 128 of the ASCII code are used for the various letters, digits and other characters that can be printed on the screen. It's time now to look at a much less obvious set of codes, the *non-printing codes*. As the name suggests, these don't produce a character on the screen, but they certainly have an effect. We can make use of them in two ways. One is by using the control (CTRL) key along with a letter key pressed at the same time, the other is by using a PRINT CHR$ instruction, with the appropriate number, 1 to 32, in brackets. The numbers follow the pattern of the letters of the alphabet for the first 26 letters, so that A = 1, B = 2 and so on. Because of this, PRINTCHR$(1) has the same effect as pressing CTRL A. The difference is that the use of the CTRL key has the effect of letting you see the result as you type the keys, but the use of CHR$ in a program is not apparent until the instruction is executed.

Fig. 4.18 illustrates the uses of these codes, as far as I can discover on my very early model of ORIC. We shall deal with many of them in detail as we go on in this book, but some of them really need a mention at the moment.

CTRL T, as we mentioned briefly earlier, toggles the upper/lower case. *Upper case*, remember, is what most of us call 'capitals', and *toggling* means switching. When you switch on your ORIC, it is arranged so that pressing a letter key always produces a capital

CTRL	CHR$	Effect
A	1	CTRL A is used in editing (early models) – see Appendix B.
B	2	–
C	3	Breaks out of a loop, stops listing from scrolling.
D	4	Double printing on or off.
E	5	–
F	6	Key sound on or off.
G	7	Sounds a 'ping'.
H	8	Cursor moves left.
I	9	Cursor moves right.
J	10	Cursor moves down.
K	11	Cursor moves up.
L	12	Clear the screen.
M	13	Produces SYNTAX ERROR if a character is present.
N	14	Clear line.
O	15	Hides screen output (on/off).
P	16	–
Q	17	Cursor on/off.
R	18	–
S	19	Hides screen output until cancelled.
T	20	Upper/lower case on/off.
U	21	–
V	22	–
W	23	–
X	24	Causes / to be printed, removes line from memory.
Y	25	–
Z	26	Black background.
ESC	27	Affects next letter typed.
]	29	Inverse video on/off.

Fig. 4.18. Using the non-printing codes with the CTRL key or by typing CHR$.

letter, and the SHIFT keys have no effect on letters, only on the keys which show more than one character. If you press CTRL T, this will switch the ORIC to print lower case letters each time you press a letter key, and capitals only when a SHIFT key is pressed at the same time. This is useful if you want messages to appear in lower case, but you need to remember that if you enter an instruction word like PRINT as print, then it will cause an error message. For that reason, I have not shown much use of lower case in the programs in this book – it's too easy to forget it until you have more experience.

Another useful feature of this code, like most of these codes, is that it can be used within a program. Fig. 4.19 illustrates a program which appears quite simple, but which switches the case of letters in line 2∅. When you type your name, you need to use the keyboard like that of a typewriter, pressing the SHIFT key down when you type the first letter of each name so as to get a capital letter.

```
5 REM CASE!
10 PRINT"UPPER/LOWER CASE"
20 PRINTCHR$(20)
30 INPUT"YOUR NAME,PLEASE-";A$
40 PRINT A$:PRINT CHR$(20)
```

Fig. 4.19. An upper-to-lower case conversion within a program.

CTRL F can also be useful if you want to use the computer quietly – it switches the key-bleep on and off. Normally, the key-bleep is on, but pressing CTRL F will switch it off. To restore the sound, press CTRL F once again. You may possibly want to do the switching in a program, in which case PRINTCHR$(6) will do the job for you.

CTRL L clears the screen, and though you could use PRINTCHR$(12) in a program, it's easier to use CLS as we have done. Another interesting code is CTRL O, which hides entry. If you type in anything after CTRL O has been pressed, it does not show on the screen, but normal service is resumed on the next line. This code cannot be used within a program so easily, but a similar effect is obtained from CTRL S, and this one works also with PRINTCHR$(19), as the example in Fig. 4.20 shows.

```
5 REM HIDE
10 CLS:PRINT
20 PRINT"PLEASE ENTER YOUR PASS CODE"
30 PRINT CHR$(19)
35 INPUT CD$
40 PRINT CHR$(19)
50 PRINT "IT WAS, IN FACT, ";CD$
```

Fig. 4.20. Hiding an input. This is useful for typing answers which you want to be concealed.

The action of CTRL D is the most difficult for the beginner to understand. When CTRL D has been pressed, everything that you type from then on appears in duplicate on the line below. This effect is used along with other instructions for creating double-height text letters, which look very effective in titles.

To show this in action, carry out the following routine:

1. Clear the screen, using CTRL L.
2. Press the space bar once to move the cursor in one space.
3. Press the down-arrow key once to move the cursor down one line.
4. Press CTRL D, then ESC, then J.
5. Now type a message – it will appear in giant characters!

Play around with this to see why you have to follow this procedure. If you start on the top line, you will get the bottom half of each letter on the top line, and the top half on the next line. If you start when the cursor is at the extreme left-hand side of the screen, you will find that the screen goes black for one line when you type ESC and J. Note that the ESC and J keys are *not* pressed together.

Now for an encore, repeat the process, but type N after ESC in place of J. Just to show that it can all be done from within a program, try Fig. 4.21. This uses PRINT instructions to place the

```
 10 CLS:PRINT:PRINT" ";
 20 PRINT CHR$(4);CHR$(27);"N THIS IS A
TEST OF ORIC"
 30 PRINTCHR$(4)
 40 END
```

Fig. 4.21. Producing double-sized characters which flash.

cursor at the correct place, so don't miss out the space between the quotes in line 1∅. Line 2∅ then puts in the control and the ESC codes (CTRL D is 4, ESC is 27), and the N which starts the title is then

Character following ESC	Effect
H	Single-height steady ordinary character set.
I	Single-height steady alternate character set.
J	Double-height steady ordinary character set.
K	Double-height steady alternate character set.
L	Single-height flashing ordinary character set.
M	Single-height flashing alternate character set.
N	Double-height flashing standard character set.
O	Double-height flashing alternate character set.

Note:
These effects continue to the end of the line that contains the ESC code unless countermanded. The ESC code and the letter code following it will require a total of two spaces on the screen.

Fig. 4.22. How different letters used after ESC (CHR$(27)) affect the appearance of the letters following – for that line only.

taken as being part of the ESC code. Because of this, it is *not* printed, but has the effect of causing double-height flashing characters. Fig. 4.22 shows some of the other letters that can be used following ESC in this way.

Chapter Five
Arrays and Files

We've used three types of variables so far, the string variables, and the real and integer number variables. None of these types allows us to cope with lists of items, however, and that's why a different type of variable, the *array*, is used. An array is a list of variables which all use the same 'name'. We distinguish the items from each other, and from a simple variable with the same name, by numbering them, using what are called subscript numbers in brackets.

Let's take as an example the variable name L$. This is an ordinary string variable name, so we can assign this name to any string. We can also have a string variable called L$(1), pronounced as EL-STRING-OF-ONE, which can also be assigned to any string value, and which is treated by the computer in every way as a different variable. What makes this style of variable so useful is that we can refer to the items of the string variable by numbers.

Take a look at the simple example of Fig. 5.1. The data consists of the days of the week. These names are read in a loop in lines 2Ø to 4Ø. In this loop, the number variable N will take values that step from 1 to 7. As each different value of N is taken, the day name is read into a variable DA$(N), so that DA$(1) is MONDAY, DA$(2) is TUESDAY, and so on. Lines 5Ø to 7Ø demonstrate that this has

```
 5 REM ARRAY
 10 CLS:PRINT
 20 FORN=1TO7
 30 READ DA$(N)
 40 NEXT
 50 INPUT"DAY NUMBER,PLEASE-:";DA
 60 IF DA>7 OR DA<1 THEN PRINT"IMPOSSIBL
E...TRY AGAIN":GOTO50
 70 PRINT DA$(DA)
 80 END
 100 DATAMONDAY,TUESDAY,WEDNESDAY,THURSD
AY,FRIDAY,SATURDAY,SUNDAY
```

Fig. 5.1. An example of an array, in this case a string array.

been done by allowing you to select a number, and then printing the day that corresponds to that number. Line 6ø is needed to reject incorrect number choices, because an incorrect selection here could cause, at best, a blank to be printed; at worst, a hangup with an error message.

Now this is fairly straightforward, but it introduces a lot of very useful new principles. The most important one is that the items of the list or array all use the same variable name, DA$ in this example. We can pick out any one item from the list simply by attaching a number to this variable name, by taking DA$(3), for example, or DA$(5). We can use another variable name, a number variable, within the brackets, as we've illustrated in line 7ø of Fig. 5.1, and we can even use an *expression*. An expression is a formula for calculating a number, like 2*N − 1, and we can have such an expression within the brackets of the variable array name.

Now look at another example, in Fig. 5.2. This is a translation

```
 5 REM GREEK TO ME
 10 CLS:PRINT:F%=0
 20 FOR N=1TO7
 30 READ DA$(N),GK$(N)
 40 NEXT
 50 INPUT"TYPE DAY IN ENGLISH,PLEASE";TD
$
 60 FORN=1TO7
 70 IF LEFT$(TD$,2)=LEFT$(DA$(N),2)THEN
110
 80 NEXT
 90 IF F%=0THEN PRINT "NOT FOUND,PLEASE
TRY AGAIN":GOTO50
 100 END
 110 PRINT"IN GREEK,THIS IS PRONOUNCED-"
 120 PRINTTAB(16)GK$(N):F%=1:GOTO80
 200 DATAMONDAY,THEF-TE-RA,TUESDAY,TREE-
TE,WEDNESDAY
 210 DATATE-TAR-TE,THURSDAY,PEMP-TE,FRID
AY
 220 DATAPA-RA-SKE-VEE,SATURDAY,SA-VA-TO
,SUNDAY,KIRI-A-VEE
```

Fig. 5.2. An English day to Greek pronunciation guide using two arrays.

program that will let you type in the name of a day (or just the first two or three letters if you like) in English, and will then give you the pronunciation of the Greek name for that day. It doesn't print the Greek *spelling* – that needs rather more advanced programming of the type that we shall meet in Chapter 9. Line 1ø contains a new departure in programming for us – a *flag*. F% is an integer that will have one of only two values in our program, ∅ or 1. A ∅ means that the program has not been able to identify the day that you typed (you got your fingers knotted!), a 1 means that a match was found. We'll see later how this is used.

In lines 3∅ to 4∅, we need the English names of the days, and the Greek pronunciations, put into two arrays, DA$ and GK$. I've picked these variable names deliberately so as to remind me of what they represent. The next section is the actual data processing part. You type the name of the day in English – you can, in fact, get away with the first two letters. Line 6∅ sets up a loop of seven items which, in line 7∅, compares the first two letters of the day name that you have entered with the first two letters of each day stored in the array DA$. Why use just the first two letters? First of all, because the first two letters are unique – no two days have identical first-two-letter sets. Secondly, because it ensures that even if you spell (or type) the rest of the day name incorrectly, the recognition part of the program will work. When line 8∅ runs, we will have checked through all the names of the days, and if none of them matches the one you typed, the 'flag' F% will still be zero. In line 9∅, then, the message will be printed.

If the two starting letters match up, however, the test in line 7∅ switches the program to line 11∅, and the item of the GK$ array is printed. By putting the pronunciations of the Greek days in exactly the same order as the names of the English days, we ensure, for example, that the second day in the English list, TUESDAY, corresponds to the second item in the Greek list, TREE-TE. This applies in the same way to all of the other items.

When a Greek pronunciation has been printed, the flag F% is changed to 1, and the program goes back to line 8∅. This ensures that the FOR...NEXT loop that started in line 2∅ finishes correctly, and when line 9∅ is executed, the message is ignored because F% is now 1, not ∅. This program is worth spending some time on and adapting to your own purposes, because it illustrates so many useful principles.

Let's take another example of an array in use. Fig. 5.3 illustrates a 'prize-draw' program. The array NM$ holds a list of names and I've left it to you to put in as many names as you want. You have to decide at the start, however, how many names you are going to use. The reason is that the ORIC, like all computers using the standard Microsoft type of BASIC, needs some notice if you are going to use an array which has subscripts greater than 1∅. You can have an array A$ which has items A$(1) to A$(1∅), ten items, or A$(∅) to

```
  5 REM DRAW!
  10 CLS:PRINT
  20 INPUT"How many names in the draw";MX
  30 DIM NM$(MX)
  40 FORJ=1TOMX
  50 PRINT"Name ";J;"-";:INPUT NM$(J)
  60 NEXT
  70 PRINT"How many prizes ";:INPUT PX
  80 FORJ=1TOPX:CH=INT(RND(1)*MX+1)
  90 PRINT"Prize No. ";J;" goes to ";NM$(
CH)
  100 MX=MX-1:REM ONE OUT
  110 A$=NM$(CH):NM$(CH)=NM$(MX+1):NM$(MX
+1)=A$
  120 NEXT
  130 END
```

Fig. 5.3. A prize draw program which prevents anyone getting more than one prize!

A$(1∅), eleven items, without any special preparation. If, however, you want to use numbers greater than 1∅, you have to prepare the ORIC to take a larger list. This is done using the DIM instruction. DIM means 'DIMension', and when we dimension an array, we instruct the computer to make space in its memory for an array of a specified name, and with a specified number of items. You don't *have* to use as many items as you specify, but you must not use more. If you attempt to use more, the computer will hang up with an error message, BAD SUBSCRIPT.

In line 3∅ of Fig. 5.3, then, we specify that the names of the people in the draw will be held in a string array NM$, and that the maximum number that we will use will be MX. We could dimension other arrays in the same line if we wanted to. If we had a number array A with up to 20 items, for example, in the same program, we could write line 3∅ as:

```
3∅ DIM NM$(MX), A(2∅)
```

using a comma to separate the items.

The next step is to read all the names that you want to use into the string array, NM$. This is done by lines 5∅ and 6∅, using a FOR...NEXT loop. After this straightforward piece of programming, we get to the interesting bit. Line 7∅ asks you how many prizes there will be, and this number is allocated to a variable name PX. Lines 8∅ to 12∅ then pick a name for each prize, and ensure that no-one gets more than one prize unless his or her name was entered more than once.

How do we do it? To start with, line 8∅ contains the

FOR...NEXT loop that controls the number of prizes that are on offer. Each time we go through this loop, a random number is picked, and given the variable name CH. The prize is then awarded in line 9ϕ. In line 1ϕϕ, the number of names on the list is reduced by 1, because 1 name has been chosen. In line 11ϕ, we take the name which got the prize and assign it to A$. A$ is just a variable name that we use temporarily when we swap this selected name with the one at the end of the list. The next two parts of line 11ϕ do that swap operation. NM$, the item we chose as NM$(CH) has its name put into NM$(MX+1) – the last item on the list. Remember that MX was reduced by 1 in line 1ϕϕ. If you started with MX=2ϕ, then MX becomes 19 in line 1ϕϕ, and line 11ϕ swaps name 2ϕ with the one that was chosen. Because the list has only 19 items to choose from, however, item 20 cannot be chosen in the next choice, which is how we prevent anyone from getting more than one prize. In the last part of line 11ϕ, the end item is made equal to A$, which is the item that was chosen.

Whatever name is picked in lines 8ϕ and 9ϕ, then, is put to the end of the list in line 11ϕ, and the list is made one item smaller. The result is that the name at the end of the list cannot be picked again. Using our example again: if we started out with 20 items, then after picking one, the chosen name becomes number 20 and the next selection is from 19 only. Each time we go through the loop between lines 8ϕ and 12ϕ, we repeat this process until all the prizes are safely allocated. It brings a bit of sparkle to the Village Hall Restoration Draw, doesn't it? The suspense will be better if you put a delay before the NEXT in line 12ϕ, and make it look as if the computer is thinking hard about the choice. As an alternative, you could have a 'press any key to proceed' step, so that you have time to announce the winners' names one by one between draws.

Matrices

A *matrix* is a form of two-dimensional array. The name is also used for a management system in which everyone has two bosses and needn't pay any attention to either. In the sense that we use the word matrix here, it's the special form of array which has more than one subscript. A two-dimensional array means that each variable name will be followed, within brackets, by two numbers separated by a comma. We can think of these numbers as representing rows and columns. The first number is the number of the row (or line); the second number is the number of the column.

Let's take an example. Suppose we are working with a translation program for days of the week. We could keep English names in one array, French names in a second array, and Spanish names in a third array, and so on. A much easier method, however, is to keep all of the names in a single matrix. Fig. 5.4 shows this in action, using the example of English, French and Spanish words.

```
 5 REM TRANSLATION
 10 CLS:PRINT:X%=0
 20 FORJ=1TO7:FORK=1TO3
 30 READ DA$(J,K):NEXT K,J
 40 PRINTTAB(14)"Translation":PRINT
 50 PRINT"Please select- French(F) or Sp
anish(S)"
 60 GETA$:IFA$="F"THENK=2
 70 IFA$="S"THENK=3
 80 IF A$<>"F"AND A$<>"S"THEN PRINT"INCO
RRECT SELECTION -TRY AGAIN":GOTO50
 90 INPUT"Day in English";DA$
 100 FORN=1TO7
 110 IF LEFT$(DA$,2)=LEFT$(DA$(N,1),2)TH
EN PRINT" translates to ";DA$(N,K):X%=1
 120 NEXT
 130 IF X%=0THEN PRINT"Not found,please
try again"
 140 X%=0:GOTO50
 1000 DATAMONDAY,LUNDI,LUNES
 1010 DATATUESDAY,MARDI,MARTES
 1020 DATAWEDNESDAY,MERCREDI,MIERCOLES
 1030 DATATHURSDAY,JEUDI,JUEVES
 1040 DATAFRIDAY,VENDREDI,VIERNES
 1050 DATASATURDAY,SAMEDI,SABADO
 1060 DATASUNDAY,DIMANCHE,DOMINGO
```

Fig. 5.4. A translation program using an array. Note that line 11Ø only *just* fits into the ORIC line. A subroutine (see later in this chapter) could be used to shorten this line.

Line 1Ø clears the screen, prints down one line, and then sets the flag variable X% to zero. The matrix is filled by lines 2Ø and 3Ø. Two loops are needed, with the rows represented by variable J and the columns by variable K. The loop that uses K is completely enclosed, or 'nested', in the loop that uses J. It's easier to see what is going on if we go through the action just once. On the first time round the J loop, the value of J is 1, and we then repeat the inner K loop three times. The first READ will allocate DA$(1,1) to MONDAY, and the NEXTK part of line 3Ø will make K = 2. The READ on the next time around will then allocate DA$(1,2) to LUNDI, and K will become 3. The last READ will allocate DA$(1,3) to LUNES, and this ends the K loop for this set of names. The NEXTJ part of line 3∅ then takes charge, making J = 2, and repeating the K loop so as to read DA$(2,1), DA$(2,2) and DA$(2,3) on the second time through the K loop. Note that line 3∅ uses NEXT K,J, which is a shortened way of writing NEXT K:NEXT J.

The rest of the program uses methods that should be familiar to you by now. Lines 4∅ to 8∅ print a title, and ask you to select your choice of language by typing F or S (no, it's *not* Fast or Slow!). Line 8∅ intercepts any incorrect answer, and the correct response is used to select a value for variable K to use in the printing of the French or Spanish words. The selection is made in lines 9∅ to 14∅. The English word parts of the matrix are compared with the word that you have requested. When the correct English word has been found, then the translation is given by printing the column of the same array row which is selected by the value of K. When a selection has been made, X% is set to 1 so that the error message in line 13∅ is not printed.

Now quite a number of improvements can be made in this program. To start with, the GOTO 5∅ in line 14∅ will cause an endless loop which isn't easy to get out of, so we need some easier way out. An obvious method is to allow the choice of Q (for Quit)

IF...THEN...ELSE gives a choice of decisions. For example:

```
6∅ GET A$
7∅ IF A$ = "F" THEN K=2 ELSE IF A$ = "S" THEN K=3
ELSE GOTO 5∅∅∅
.................................
5∅∅∅PRINT "INCORRECT SELECTION - PLEASE TRY
AGAIN":GOTO 6∅
```

REPEAT UNTIL allows a loop to be used which, unlike the FOR...NEXT loop does not depend on a count. For example:

```
45 REPEAT
.........................
14∅ X%=∅
15∅ PRINT"PRESS Q TO STOP, ANY OTHER
KEY TO REPEAT"
16∅ GET A$
17∅ UNTIL A$ = "Q"
18∅ END
```

Fig. 5.5. Useful instructions which are promised for the later versions of the ORIC.

as well as F or S. An extra line like 75 IF A$ = "Q" THEN END will serve you well. Another point is the rather clumsy programming of tests in lines 6∅ to 8∅. On my early ORIC, this was unavoidable, but the later versions will have two extensions to the BASIC language which will considerably improve the planning of programs. One of these is IF...THEN...ELSE, which allows an option to the IF...THEN test. Another improvement is the availability of the REPEAT...UNTIL loop, which will keep a loop repeating until some terminator, such as Q, is typed. The advantage of this type of loop as compared to a FOR...NEXT loop, is that it isn't tied to a definite number of repetitions. Fig. 5.5 shows how these commands could be incorporated into the program. I must stress that I could not try these out on my early ORIC.

Subroutines

In any program, you may find that you have to type similar lines over and over again. For example, you may want to use a GET A$ and a test of quantities got by GET, several times. A problem like this is best tackled by a *subroutine*, which is a section of a program that can be 'called up' to be used from any part of another program. Take a look at an example, in Fig. 5.6. You might call this a Yellow Subroutine – but it does at least point out how we make use of a

```
 5 REM SUBROUTINE
 10 PRINTTAB(14)"SUBROUTINES"
 20 PRINT"THIS IS A";
 30 GOSUB1000
 40 PRINT"SUBROUTINE":PRINT
 50 PRINT"IT COLOURS EVERYTHING";:GOSUB1
000:PRINT"NO MATTER
 60 PRINT"WHERE IT IS PLACED."
 70 END
 1000 PRINT" *YELLOW* "
 1010 RETURN
```

Fig. 5.6. Using a subroutine. The subroutine can be called from any part of a program, and will return to the part immediately following.

subroutine. The subroutine is very simple – it consists of an instruction to print the word YELLOW with asterisks around it. This PRINT instruction is in line 1∅∅∅, and line 1∅1∅ contains the instruction RETURN. This means that the program must resume at a place immediately following the point where the subroutine was called. The calling instruction is GOSUB, followed by the number of the line in which the subroutine starts. Subroutines should be placed so that their line numbers are higher than the line which carries the END instruction. If this is not done, there is a chance that the computer will carry out a subroutine without having come to a GOSUB instruction. This will cause a hang-up, with the message RETURN WITHOUT GOSUB displayed.

Now for a closer look. In line 2∅, the words THIS IS A will be printed, and the semicolon prevents a new line from being taken. Calling the subroutine in line 3∅ causes the word *YELLOW* to be printed following the A and the semicolon here once again keeps the printing on the same line. The RETURN in line 1∅1∅ then causes the program to return to line 4∅ and print the word SUBROUTINE. Line 5∅ then demonstrates that a subroutine can be called from the middle of a multistatement line, providing that we separate the statements correctly by using colons.

This is a very trivial example, but the principle is very important. First of all, it allows us to type frequently used bits of programs once and once only. We can then put GOSUB ... to whatever line number starts the subroutine whenever we want to use it, and the program will automatically resume when the RETURN is encountered. The second advantage of GOSUB is that it can often be used to replace GOTO. When you use GOTO to shift to another line, you can't be sure where you will resume – you will need to use another GOTO to achieve a return. A GOSUB will always come back to the instruction following the GOSUB, and that's a very great advantage. In general, the fewer GOTO's we have in a program, the easier it is to write and to understand. The third advantage, which we shall look at in more detail in Chapter 6, is that GOSUB can make the planning of a program very much easier.

```
  10 T$="SUBROUTINE TEST"
  20 GOSUB 1000
  30 PRINT:PRINT"PICK A NUMBER"
  40 GOSUB 2000
  50 T$="THE NUMBER WAS "+A$
  60 GOSUB1000
  70 END
1000 PRINTTAB(20-LEN(T$)/2);T$
1010 RETURN
2000   PRINT"TYPE NUMBER 1-5":GET A$
2010 V=VAL(A$)
2020 IF V<1 OR V>5 THEN PRINT"INCORRECT
NUMBER-TRY AGAIN":GOTO2000
2030 RETURN
```

Fig. 5.7. Using subroutines in a short program.

Now take a look at a second example, Fig. 5.7. This one uses two subroutines, in lines 1ØØØ and 2ØØØ, one to print, the other for input. The subroutine in line 1ØØØ causes a string T$ to be printed centred on a line of the screen. Providing that the line is not too long (38 characters maximum), this can be used for centring titles. A long title could be broken into sections, each section equated to T$, and the subroutine called. The subroutine in line 2ØØØ uses GET A$ to detect a key being pressed, and then tests the result to check that it is within the range that was asked for. In line 6Ø, the result of this can be printed, centred once again on the line by using the first subroutine. Note the use of concatenation to combine the strings.

Menu choice

Many programs of the data processing type are *menu-driven*. This means that at various stages in the program you are presented with a list of choices and asked to select from the list by pressing a number or letter key. Fig. 5.8 shows an example of a typical screen display of a menu. In this example, there are four choices, and the choice is made by tapping a number key. The instructions state that it is not necessary to press RETURN, as the choice is made by using GET rather than INPUT.

```
              MENU
    1. ENTER ITEMS.
    2. SORT INTO ORDER.
    3. RECORD ON TAPE.
    4. REPLAY FROM TAPE.
    5. END PROGRAM.
```

Fig. 5.8. How a typical menu looks on the screen.

Fig. 5.9 shows a piece of program that would lead to a menu selection. It makes extensive use of GOSUB rather than GOTO for reasons that have been explained previously and also introduces a few new ideas. We start by clearing the screen and using a subroutine to print the word MENU centred on the screen line. The centring routine will probably be used many times, so that a subroutine is particularly appropriate here. The menu choices are printed in lines 4Ø to 8Ø. These are imaginary items – the program does not deal with them – but they are typical of a data processing type of program.

```
 5 REM MENU
 10 CLS:PRINT
 20 T$="MENU":GOSUB500
 30 PRINT
 40 PRINT"1. ENTER ITEMS."
 50 PRINT"2. SORT INTO ORDER."
 60 PRINT"3. RECORD ON TAPE."
 70 PRINT"4. REPLAY FROM TAPE."
 80 PRINT"5. END PROGRAM."
 90 GOSUB600
 100 ON V GOSUB1000,2000,3000,4000,5000
 110 PRINT"DO YOU WANT TO RETURN TO THE
MENU?"
 120 GOSUB800
 130 IF A$="N"THEN END
 140 GOTO20
 500 PRINTTAB(20-LEN(T$)/2)T$:RETURN
 600 GET A$:V=VAL(A$)
 610 IFV<1 OR V>5 THEN GOSUB700:GOSUB600
 620 RETURN
 700 PRINT"Impossible choice"
 710 PRINT"Please try again"
 720 PRINT"Range is 1 to 5 only..."
 740 RETURN
 800 GET A$
 810 IF A$<>"N"ANDA$<>"Y"THENPRINT"Y OR
N,PLEASE":GOSUB800
 820 RETURN
 1000 PRINT"ITEM 1":RETURN
 2000 PRINT"ITEM 2":RETURN
 3000 PRINT"ITEM 3":RETURN
 4000 PRINT"ITEM 4":RETURN
 5000 END
```

Fig. 5.9. A typical 'menu-driven' program. The subroutines starting at line 1ØØØ have not been written in full – there isn't room in this chapter!

The GOSUB6ØØ in line 9Ø uses a GET A$ to detect any key being pressed. The number value of this key is also found by using VAL(A$), so that in line 61Ø, V should have a value between 1 and 5 if you pressed one of the permitted number keys. If you pressed any other key, however, V will be zero or greater than 5, and this is detected in line 61Ø. The GOSUB7ØØ then leads to a printed explanation for the rejection of your choice, and the GOSUB6ØØ in the same line gives you another attempt to get it right. This is an example of a subroutine calling itself, a technique that is called *recursion.* When you press a key that gives an acceptable number, the subroutine returns you to line 1ØØ, which is where the choice is made.

The ON V GOSUB instruction means 'select from a list the correct number for the item'. If V = 1, then the first subroutine number is used; this is 1ØØØ so that the subroutine at line 1ØØØ will be run. In this example, the subroutine simply prints ITEM 1 to show you that it was selected. If V = 5, then the fifth number in the list is used, causing the subroutine at line 5ØØØ to be used, and the same applies to the others. It isn't necessary to use subroutine line numbers like 1ØØØ, 2ØØØ ... 5ØØØ for numbers 1 to 5. We could just as easily use 7ØØØ,21Ø,52Ø,3ØØ,4ØØØ, because it's the *order* that counts, not the actual numbers. All that you have to do is to ensure that there is a subroutine starting at each of these line numbers.

There is a similar instruction, ON N GOTO which acts in the same 'select-from-a-list' way as the GOSUB example that we have used. If the program ends after each selection, there is nothing wrong in using GOTO, but for any other purposes ON N GOSUB is better. 'N' in this description just means the number variable that we use: we have used V in the example. Note, by the way, that a menu should always have a 'quit' option, one that will end the program. This avoids the problems of having to type a lot of useless data just to get a program to stop!

Data recording and replaying

Computers generally have special instructions for the recording and replaying of variables, including arrays and matrices, separate from the recording and replaying of programs. It's a very great advantage to be able to record and replay variables in this way. You can use one program in which data is gathered, for example, and record the data that this program gathers. This data can then be used by any other program that you care to write. This facility allows you to split what might be excessively long programs into several parts, all using the same data – it's particularly useful if you want to do a lot of data processing with the 16K machine. In addition, once you have recorded data, you can change the program that recorded the data, and then load the data in again. If you attempt to make any changes to a program while data is stored, all of the data will be lost.

As it comes, the ORIC does not directly provide for saving or loading data. The reason is that a cassette program will be available later which provides this facility, along with many other data processing activities. This relieves you from having to write many difficult and tedious routines for yourself, and makes the ORIC eminently suitable for all kinds of data processing work.

Chapter Six

Rolling Your Own

Until now, we have been looking at *coding* – converting the instructions that we want to carry out into the form of BASIC lines. Though you might not think so at first, coding is by far the easiest part of programming your computer. The difficult part, and the part which is most interesting, is planning a program so that it does something useful for us.

Now different users need different types of programs, but as it happens, the steps of planning a program are pretty much the same no matter what type of program you are designing. Don't be put off the job of designing your own programs by some of the textbooks which show spider-web diagrams that are supposed to simplify things. All you need to design a program is a lot of paper, and a problem to solve.

Whichever type of program you want to write, the most difficult part is making a start. The best start you can make is to write a description of what you want the program to do. If it's a business or other 'data processing' type of program, then write down what data it needs and what it is supposed to do with that data (not necessarily in that order!). If it's a game of any kind, then write down what the rules of the game are to be. Inventing an original game is one of the hardest things there is, which is why so many computer games look so similar. Take your time over the outline stage – you may find that you have a lot of second (third, fourth ...) thoughts once you see your ideas on paper.

Now that you have an outline of what you want to do, start thinking in computer terms. The computer can print up to 25 or so lines of items on the screen, it can request an input from you, it can print on paper, record on tape, replay from tape. At what stages in your program do you need an output on the screen? If you are designing a game, you may want the screen to be giving you an output almost constantly, and you will have to start to think what shapes you expect to see. Chapters 7 to 9 will then be of considerable interest to you. For data processing, you may only want the screen used to print a reminder or a reply to a single question. It's more usual in data processing programs for the paper printer to be used for most of the output rather than the screen. This paper output is called *hard copy*.

At this point, you can start to make a list of what you expect to see on the screen at various stages in the program. Such a list for a small data processing program is illustrated in Fig. 6.1. You also need a list of what inputs are expected. In a games program, you might have to

1. Main menu, with note on how to choose.
2. Other menus for selecting sorting methods.
3. Prompts for input of data.
4. Warnings to prepare cassette for recording/replayed.
5. Reminder when data has been recorded/replayed.
6. Warning if running out of memory (use FRE instruction).
7. Reminder if incorrect form of data entered.
8. Review of newly entered data.
9. Review of all, or selected, data.
10. Reminder to switch on printer.
11. Reminder while sorting is being done.
12. Reminder when program has ended.

Fig. 6.1. A list of what you expect to see on the screen – an essential early stage in program planning.

select how many magic amulets, scarabs, poison darts you can take with you to the Caverns of OR–IK. You might, on the other hand, want to use just the arrowed keys to cause a gun to be aimed, or a spaceship steered. In a data processing program, you might want to enter names, addresses, ages, tax codes, number of children, and so on. Whatever your application, there will almost certainly be some stage in the program where you have to enter data. If you plan where these entries are to be made, and show *what* entries have to be made, it makes the program planning much easier. Fig. 6.2 shows an

1. Menu selection, main and minor.
2. "Press any key" before record/replay/print stages.
3. Main input of data.
4. Correction of data.
5. Specify how items are to be selected or sorted.
6. YES/NO inputs for choices.

Fig. 6.2. Entries into a program. Listing these is another essential preliminary.

example. Finally, you have to decide how the program ends. For any games program, your outline scheme should have included some description of how a score is made, and you have to decide what score or other event will end the program. You may decide that a score of 10000, or a hit by an Astatine Bomb is the finishing event, but this has to be written down so that it can be programmed. For a data processing program, a 'quit' option in the menu is the usual way of terminating the program.

The next step is to write the outline in the form of stages. These should be main sections – not under any circumstances detailed lines of program. Fig. 6.3 shows what these might look like for a typical data processing program and for a simple game. What you now have

Main menu	Rules
Data entry	Start play
Record data	Score
Replay data	Test for end
Select data	Offer another run
(a)	(b)

Fig. 6.3. Outlines of programs: (a) data processing, (b) games.

is some sort of structure, a foundation on which you can build your programming. The important point is that you now know which piece of program follows another; the 'flow' of the program. This is the time for second thoughts, improvements, and spotting repetitions of the sort that positively beg for the use of a subroutine.

This system of designing programs is called 'top-down' programming. It consists of designing the main outline first and, in the course of several more steps, filling in the details. The most significant advantage of this method is that you know where you are going at all times, and you can arrange things so that the program is easy to write, easy to test, and easy to change if anything has to be changed. The use of top-down design is particularly assisted by writing program actions as subroutines, as we shall illustrate by examples.

Designing a data processing program

An example is always an easier way of showing what has to be done than a set of rules. Let's suppose that you want a program that will store data on your friends. You want to store names and addresses, of course. It might be a good idea to keep a note of the telephone numbers as well. If we also store their date of birth, we may be able to get the computer to remind us about birthdays. Finally, if we keep a note of how far away they live, we can get a good idea at any time of how easy or difficult it would be to get them all to a party.

The important point here is that the computer isn't magical. It can make use of information that you provide it with. The data we've imagined here it could be programmed to pick out all the friends with a birthday in May, all the friends who live within four miles from you, all the friends whose surnames start with J, and so on. It couldn't tell you which friends are left-handed and which are blonde because you haven't put that information in.

We start, then, by listing what we expect this program to do, as illustrated by Fig. 6.4. This is a very brief description, but it contains all that we have to do to put information into the computer and to

```
ENTER NAME, ADDRESS, PHONE NUMBER, DATE OF BIRTH,
  DISTANCE
STORE DATA
RECALL DATA
SELECT BY NAME, MONTH OF BIRTH, DISTANCE
UPDATE OR CHANGE DATA
```

Fig. 6.4. A brief description of our example of a data processing program.

make use of it. We shall need to be able to enter data, and the list shows what items we expect to enter. We shall need to be able to store this data, separate from the program, on tape or on disc, and to recall (replay) the data when we need it. We need to be able to process the data. The processes that appear here are selecting by name, by month of birth and by distance. We can therefore find the details for anyone whose name we type, or find the names of anyone with a birthday in June, or find the names of all the people on the list who are up to ten miles away from us. This 'data processing' does nothing that a good card-index file would not do (and you never see people demonstrating against card index files!), but the computer can do the job a lot faster. Finally, we need to be able to update the data file. Friends move house, fall out of favour, we meet new friends, and so on, making it necessary to record a new data file at intervals. Your program should make it possible to carry out these changes without having to type in all the names and data again.

The next step in planning is deciding what you need to see and enter at various stages. This needs a lot more detail, and Fig. 6.5 shows what you might need for this program. You can see that there is a lot more detail here – we are building flesh on the skeleton of Fig. 6.4, and we need all of this before we can start to think about programming.

ENTRY: Type name, address, phone number, date of birth, distance. Screen should show prompt for each item. Prompt must show form of answer (like ∅3–∅5–65 for date of birth). Show complete entry, and ask if changes are needed before putting into array.

STORE: Screen message "PREPARE CASSETTE FOR RECORDING". Then use "PRESS ANY KEY" step to give time.

RECALL: Screen message "PREPARE CASSETTE FOR REPLAY". Use "PRESS ANY KEY" to proceed. *Note* – can assign "PREPARE CASSETTE FOR " to one string, and just add RECORDING or REPLAY in the subroutine.

SELECT: Menu needed to choose item to be selected.

UPDATE: Menu needed for new name, change details of old name, and then which detail choice.

Fig. 6.5. Filling in more detail for the program.

Take a look at some of this detail, because it illustrates the way in which we convert ideas into program lines. On the ENTRY section, for example, the detail is a reminder that we need a prompt for each entry. A 'prompt' is a message on the screen to remind you of something, and the prompts here are intended to remind you that items such as dates have to be entered in a way that suits the program. For example, entering a date as ∅3–∅5–65 makes it very much easier to pick out the month from a date – in this case by using MID$(DT$,4,2) where DT$ is the variable name that is used to store the date. You will have to give careful consideration to any other items that you are going to use for selection. If you are selecting by surname, for example, the name should be stored with the surname first. If the program can't find the surname that you ask for (which might be because of your typing), then it can be arranged to print all the names that start with the same three letters (or two, or whatever you like).

Once you have considered the plan in this much detail, you can start planning the program itself. This will obviously be a program that will use a menu, and you can now write the menu that you expect to see on the screen when you start the program. Don't write the program for the menu yet – it's too early! Fig. 6.6 shows what you might expect of the menu for this program.

```
MENU

1. ENTRY OF NEW DATA.
2. STORE ON TAPE.
3. REPLAY FROM TAPE.
4. SELECT DATA.
5. UPDATE DATA.
6. END.
```

Fig. 6.6. The menu for the example shown.

Now comes the crunch. We can now design a 'core' program which is centred around this main menu. It will start by clearing the screen and asking you if you want any help in using the program. The lines of code could read:

```
PRINT"DO YOU NEED ANY HELP? (Y/N)"
GOSUB5ØØ: IF A$ = "Y" THEN GOSUB 6ØØ
```

The key to good design is in the use of these subroutines. The subroutine at 6ØØ is one which will accept a Y or a N answer, and reject any others. It 'passes back a value' in the form of the variable A$, which will be either Y or N. If the Y answer is received, then another subroutine, starting at line 6ØØ, is called. This should present the instructions in a reasonably brief form. If the instructions need more than a full screen, you will have to print a set of lines, and then use a 'press any key' step so that the user can see the rest of the instructions after having read the first section. In general, I much prefer to have lengthy instructions on paper, where they can be read separate from the computer. Computer instructions should be a brief reminder only.

The point of putting in a subroutine here is that it allows you to defer the effort of writing the instructions until the program is working as you want it. That's the essence of this approach to design – leave detail until last. The next step in this program would be to clear the screen and display the main menu.

What next? Just as we illustrated in Chapter 5, we follow the menu with a subroutine which lets us input a number. The number will have to be tested for range – you can't use Ø, or numbers greater than 6 in this example, and by using GET we prevent any entering of fractions like 3.6. Once the number is checked, it can be used in an ON N GOSUB type of step to direct each menu choice to a suitable subroutine.

Following this, the last step is the question 'DO YOU WANT TO RETURN TO THE MENU?(Y/N)'. We already have a subroutine to deal with Y or N answers so we can use this again. If the answer is Y, then the GOTO step will send the program back to the start of the menu again – an alternative for the later model of ORIC will be UNTIL A$ = N, with the REPEAT instruction placed just before the menu lines. Fig. 6.7 shows a core program which is designed along the lines we have discussed.

Now this example illustrates all the design principles that we need! Each subroutine now has to be written. Some of them, like printing a

```
 1 REM DATA PROCESSING
 2 REM USE THESE LINES FOR DIM
 3 REM AND OTHER PREPARATORY
 4 REM STATEMENTS ********************
 5 REM YOU WILL HAVE TO WRITE THE
 6 REM SUBROUTINES BEFORE THIS CAN
 7 REM BE USED ******************
 10 CLS
 20 TT$="FRIENDS"
 30 GOSUB800:REM CENTRE TITLE
 40 PRINT:PRINT
 50 PRINT"DO YOU NEED ANY HELP?(Y/N)"
 60 GOSUB500:REM Y/N TEST
 70 IF A$="Y" THEN GOSUB600:REM INSTRUCT
IONS
 80 WAIT100
 90 CLS:PRINT
 100 TT$="MENU"
 110 GOSUB800
 120 PRINT:PRINT
 130 PRINT"1. ENTRY OF DATA."
 140 PRINT"2. STORE ON TAPE."
 150 PRINT"3. REPLAY FROM TAPE."
 160 PRINT"4. SELECT DATA."
 170 PRINT"5. UPDATE DATA."
 180 PRINT"6. END PROGRAM."
 190 GOSUB900:REM PLEASE SELECT MESSAGE
 200 GOSUB950:REM CHOICE OF NUMBER
 210 ON V GOSUB1000,2000,3000,4000,5000,
6000
 220 PRINT"DO YOU WANT TO RETURN TO THE
MENU?"
 230 GOSUB500
 240 IFA$="N"THEN END
 250 GOTO100
```

Fig. 6.7. The 'core' program illustrated.

string centred, getting a Y/N answer, choosing a number, are standard items that we have illustrated previously and which you will use in practically every program. Others, like the subroutines that deal with the menu, will have to be written, but should follow familiar lines. Finally, we have the main subroutines. Each of these should be designed in exactly the same way as we have approached the design of the main section of the program. Once more we outline what the subroutine should do, what you expect to see and enter. Once more, we can make use of other subroutines, always leaving detail until the end. As with so many other activities, the first time is the most difficult.

Games programs

The design of games programs is by no means so straightforward as the design of data processing programs. You must decide right at the beginning whether the game is to be of the 'words on the screen' variety, or whether it will include drawing – *graphics* as we call it. The principles of design are the same, but it's less likely that a game will be based on a menu. A game, in general, will consist of:

(a) Instructions
(b) Play
(c) Score
(d) Endpoint.

Your planning should consider what the rules of the game will be, what the play consists of (shooting down aliens, guessing a name, rescuing climbers, catching rabid dogs ...) and how points will be scored. You will also have to decide what events will allow play to be repeated or continued, and which will cause the game to be ended.

From there on, the steps start to look familiar. You design your 'core' games program, consisting of a set of subroutines with some decision steps. One subroutine causes the instructions to appear, one generates the screen pattern (the Galaxy, the farmyard, Sunset Boulevard, the desert, or whatever), one causes the movement (alien dropping, car moving, gun pointing ...), and another is used to keep a score when something happens.

A particularly good guide to this subject is to look at the books of games by James, Gee and Ewbank (published by Granada). By the time this book appears, an *Oric Book of Games* by these authors will probably be on its way. The previous books in the series show various types of games, with a lot of detail about how the program was designed, and what each subroutine does. A close study of these games is essential reading if you want to roll your own!

Entering, testing and fault-finding

When a program has been designed in the 'top-down' method that is so widely used, we can enter it into the memory of the computer and test it in the same sort of way. That means one piece at a time, going from the core to the least important subroutine, in the case of a typical data processing program. For the data processing program of our example, we might start by entering the core program that was illustrated in Fig. 6.7. Now when this is entered as it is, you can't run it right away because the computer will hang up with an error message when there is a GOSUB to a line that doesn't exist. As we saw in Chapter 5, though, you can type in the shorter subroutines, like the PRESS ANY KEY TO PROCEED and the centre-title subroutines, and for the rest, simply put a short PRINT item like THIS WAS CHOICE 1, and then a RETURN. When you do this, you can test the core until you are sure that it will do all that you want. Testing means that you try every option. Where there is a Y or N, you try Y, then N, and then some other key. When you have a menu choice of numbers 1 to 6, try ∅, 7, and if the choice uses INPUT, try 3.5 as well. Do, in other words, everything that a rookie user might be expected to do. If your program is good, it will refuse to be corrupted by these incorrect answers, and it will always give guidance on what went wrong, and what is expected.

Once you have the core program running really smoothly, you should record it at the start of a fresh C-30, C-60, or C-90 cassette. This is your development tape, and each stage in the development of the program should be recorded as soon as it has been tested, perhaps even before. In that way, if some disaster, like a complete hang-up, wipes out your program, you will still have a very recent version on tape to load back in.

The next step after testing and recording the core program is to get to work on the menu subroutines. You can put them into the machine in any order, but my preference is to put the data recording and replay routines in first of all. One reason is that these are usually pretty standard – when you've seen one, you've seen the lot. The other more important reason is that when you come to test the program with data, you don't want to have to type a lot of data for test purposes more than once. If your data record and replay routines work, then you can record any data that you put in for testing purposes and rely on being able to get it back into the machine later if it is needed.

Next in the line of importance comes the subroutine that is used for the entry of data. This will be a long subroutine, and it will have to call several other subroutines dealing with items such as instructing you how to present the data, checking that the data is correctly presented, making each complete set of items appear on the screen, and allowing you to make corrections. Practically all data processing programs will use a matrix or a set of arrays to hold the information, so that reviewing and correcting can be reasonably easy for any item number, and the whole list of items can be checked by making use of a FOR...NEXT loop.

Each subroutine should be checked as soon as it is written and entered, and the program, as it exists, should be recorded every time a major subroutine has been added. This way, you can be fairly certain that if the program stops working at any time it will probably be because of a new section that you have just added, and you can go back to your previous version. If you attempt to write the program in one large chunk, then the task of finding errors is a massive and very difficult one.

Testing begins in earnest when the program seems to be running properly, with no more syntax errors appearing on a run through. The most difficult testing problem is when you need to make thorough tests of a program that is intended to handle a large amount of data. The routines will have been tested as you have entered them, but only for major flaws. What you now have to find out is whether the program will work when a large amount of data is put into it. This is why it's important to have the cassette (or disc) record and replay of subroutines working well. If you have reasonable confidence in the program, and you should, then the best test data to use is the data that you actually want to use with the program. Now if anything goes wrong, and the computer stops with an error message, you can type GOTO , putting in the line number at which the recording routine starts, and be sure of recording all your data. There will be a RETURN without GOSUB error message at the end of the routine, but your data will be safe by that time. In this way, you don't have the heart-breaking task of typing it all in again.

Why should data saving be so important? The answer is that whenever you type RUN and RETURN or when you make any change, however small, in a program line, or add or delete a line, all variable values are reset to zero. All the data that you may have spent an hour typing in will be lost – unless you have recorded it. The moral is – take care of your data!

Fault-finding

Every second blue moon in the thirteenth month of the year, a big program runs exactly according to plan. If the program has been designed in sections as we've described in this chapter, then many of the faults will have been sorted out during the design and the entering of the program, and there should be few serious faults. Every now and again, though, something happens which is very difficult to trace.

The first aim of fault-finding must be to find which section of the program causes the fault. If the fault is simply a misplaced PRINT, it should be easy to find and to correct. The most awkward faults are those which cause programs to run well – as you think – but produce incorrect results. What's worse is that the incorrect results may not be spotted during testing, particularly if the program is a complicated one. This can be very serious if, for example, the program is designed to work out the income tax for a large number of people. To find such faults, you need to write, at a fairly early stage in the design of the program, a set of test data that you can use. You will have to go through the painful process of working out for yourself what answers the computer should produce with this 'dummy' data, and you need to keep a note of these answers. If you use this data to check out the program, and the program produces the answers that you expect, then the program is probably correct – at least it's doing what you expect it to do. If you find, however, that the computer results are noticeably different, then some investigation is needed. A small difference can often be caused by the 'rounding errors' that we have looked at earlier, but the major errors need more thorough investigation.

The main weapon that you have in this fault-finding is the word STOP. When you put STOP into a program line – preferably in an 'odd' line somewhere between instruction lines, the program will stop there. When the program stops, all variable values are preserved. If you have variables A$,N,J,K,Z,P in your program at that point, then you can print the values of each of these variables by using direct commands, such as PRINT A$. Once you have printed out all the variable values, then, providing you have not altered any line of the program, you can type CONT then the RETURN key, and your program will continue. You can then sneer at owners of a machine costing four times as much which can't do this!

By placing one or more STOP lines at points where you suspect that the program is going wrong, you can find out what the values of variables are each time a STOP is encountered. This applies particularly to a loop, and if you need to trace values several times, it may be useful to put your PRINT instructions into a subroutine such as:

```
30000 PRINT A$,B$
30010 PRINT C,D
30020 GET ZZ$
30030 RETURN
```

rather than using STOP. The line you now have to add to your program will be GOSUB 30000 rather than STOP. Don't forget to remove all the STOP and GOSUB 30000 instructions after you have repaired the problems!

There's one action that's better than all the advice and all the reading, though. That is to write a program of your own to solve your own problems on your own ORIC!

Chapter Seven
Graphics (I)

Graphics means printing shapes on the screen so as to form drawings rather than letters or numbers. We refer to the graphics of computers as being of two types – *low resolution* or *high resolution*. Low resolution graphics means that we work with shapes made from fairly large units called *pixels* (picture elements). High resolution graphics means that we can make up pictures using very small dots as elements, and most modern computers allow these dots or pixels to be in any one of a range of colours.

We'll start by taking a look at the low resolution graphics of the ORIC-1. These use a pattern of small rectangles that will be familiar to anyone who has used teletext (CEEFAX, ORACLE or PRESTEL), and the basic pattern shape is illustrated in Fig. 7.1. The full rectangle is the same size on the screen as a capital letter, but

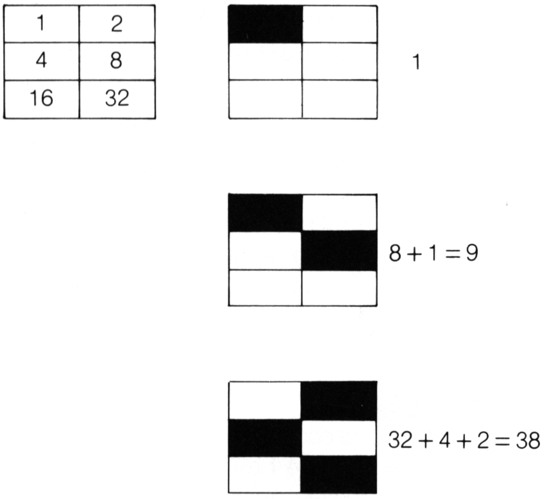
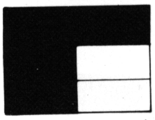


Fig. 7.1. The low-resolution graphics block. Note how each of the six sections has a reference or code number.

it is divided into six sections, and we can 'shade' or colour any one of these sections or any combination of these sections. We do this by specifying a number – you never get very far away from numbers as far as computers are concerned! Fig. 7.1 also shows what these numbers are, and how a number-code can correspond to a shape. To find what number-code you need for a shape, just put a piece of tracing paper over the basic rectangle shape in Fig. 7.1 and shade in the sections that you want to see shaded or coloured on the screen. Now add up the numbers in the sections you have shaded, and you have the code number for the graphics pattern that you have shaded.

It's not quite as simple as that, however. The number that you get from this action can't just be typed as a number or even as a CHR$ (number) because the computer will not automatically treat such numbers as being graphics shapes. What we need to do is to put in a series of codes that will instruct the computer to *prepare* for a number that is a graphics code. The whole difficulty stems from the fact that small computers can use only a small range of number codes, ∅ to 255, and everything that needs to be coded has to use this range of numbers. If that means we sometimes have to follow a rather complicated sequence of actions, then so be it.

To look at the effect of our graphics shape on the screen means that we have to learn methods of displaying it. One method is just to type the characters in! Try this: clear the screen by using CTRL L, and then space down one line (using the down-arrow key) and across one space (using the spacebar). Now tap the ESC key followed by the letter I. Nothing appears on the screen. Now type A. What appears is *not* an A, but a graphics character, and you will get a graphics character for each key you type except for the RETURN, SHIFT, DEL and cursor (arrowed) keys. The problem now is to find which key produces which character, and this brings us back to ASCII codes. If you take your graphics shape number, add 32 to it, and look up the key whose ASCII code this is, that's it! For example, consider the shape in Fig. 7.2, which has the code number 23.



1 + 2 + 4 + 16 = 23
code for shape

Add 32 to code to get 55 which is ASCII code for '7'. We can use CTRL 7 *keys* or, CHR$(55) (within a program).

Fig. 7.2. Finding ASCII codes for the block numbers.

Adding 32 to this gives us 55, and 55 is the ASCII code for the digit '7'. By pressing the '7' key, after having pressed ESC and I, we should get the shape – and we do. You must remember, though, that you have to use the spacebar to place the cursor in from the left-hand side of the screen before you use ESC I. If you do not do this, then your character will be printed in black on a black background. A modern art gallery might like the idea, but it doesn't exactly make the character visible. This applies only when you are working with the normal white background (paper) and black printing (ink). If you reverse these by using CTRL] or if you use differently coloured background and print colours, then things are different. We'll look at that point later.

Take a look now at how we draw a pattern on the screen using these graphics shapes. Let's consider the simple pattern in Fig. 7.3,

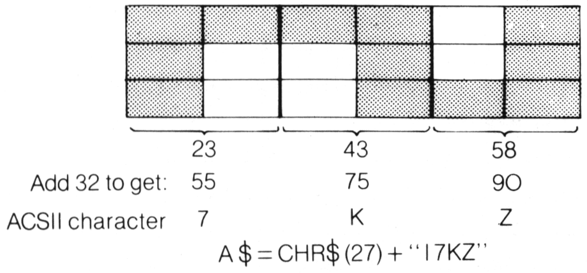


Fig. 7.3. A simple pattern using three graphics blocks (A$ = CHR$(27) + "I7KZ").

which uses three of the complete blocks. We can find the code numbers for each of these graphics blocks, and add 32 to each number so as to obtain the ASCII codes. We can then put the shapes on the screen by using ESC I, followed by the letters 7 K Z, the keys which have to be pressed in order to create the pattern.

That's a quick and convenient way of checking what the shape will be, but this isn't usually the way we want to use these shapes. We normally want to make use of graphics shapes within programs, so that we can produce heavy underlining, boxes, aliens and other patterns during the course of a program. ORIC offers a neat and simple way of doing this once you have discovered what shapes you want to string together.

Try this. Clear the screen and type:

```
A$ = " "+CHR$(27)+"I7KZ"
```

and then press the RETURN key. This puts the string which contains one space, the escape code (27), the I, and the letters that we need for the shape into one string, A$. When we print this now, using PRINT A$, we will get the graphics pattern. Try it!

Now for something more ambitious. We can make a more interesting pattern if our blocks can be stacked vertically as well as laid horizontally. Fig. 7.4 shows a 'space invader' type of shape

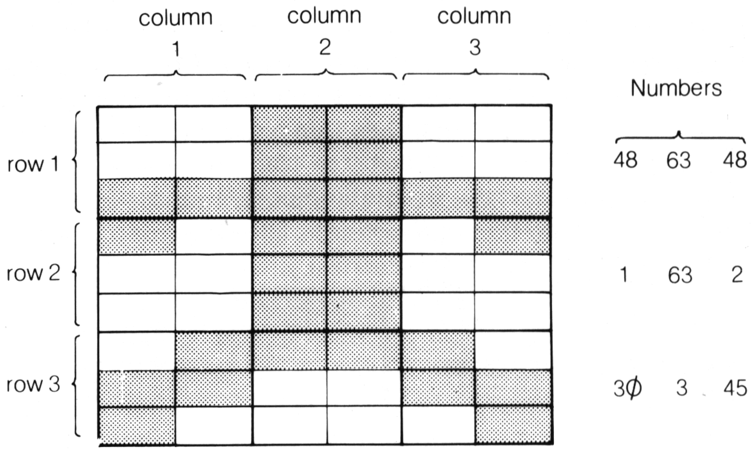


Fig. 7.4. A 9-block pattern for a 'space-invader'.

which uses nine blocks, three across and three deep. We can create a shape like this by printing three strings, one for each row of the block. Each string will have to contain three codes following the space, escape and I codes. Fig. 7.5 is a program that will print this

```
5 REM INVADER
7 A$=""
10 FORN=1TO6
20 READ J
30 A$=A$+CHR$(J):NEXT
35 CLS:PRINT
40 PRINT A$
45 A$=""
50 FOR N=1TO6:READJ
60 A$=A$+CHR$(J):NEXT
70 PRINT A$
75 A$=""
80 FORN=1TO6:READJ
90 A$=A$+CHR$(J):NEXT
100 PRINT A$
200 DATA32,27,73,80,95,80
210 DATA32,27,73,33,95,34
220 DATA32,27,73,62,35,77
```

Fig. 7.5. A program that will print the 'alien' shape.

shape. The shape looks rather lopsided in my early-model ORIC because the sizes of the pixels that make up the blocks are not identical for some reason. We have created the shape in three lines of printing, starting each line with the codes that specify graphics, and then continuing with the numbers that create the shapes. We can then place this shape on the screen where we want it by using the same TAB number for each line of print.

This is rather clumsy, though, and we would prefer to be able to print this alien shape just by printing one string, A$ in this example. The program of Fig. 7.6 allows you to do just this. Each line of data

```
5 REM ALIEN2
10 A$=""
20 FORN=1TO3
30 FORX=1TO6
40 READJ:A$=A$+CHR$(J):NEXT
50 A$=A$+CHR$(10)+CHR$(13)
60 NEXT
70 PRINT A$
100 DATA32,27,73,80,95,80
110 DATA32,27,73,33,95,34
120 DATA32,27,73,62,35,77
```

Fig. 7.6. Packing the data into one string.

places the codes for three blocks of the shape into the string A$, then line 5∅ adds the line feed (1∅) and carriage return (13) codes so that a new line will be selected. Another set of codes is then added, another line feed and carriage return, then another set of codes. This has the effect of causing three sets of blocks to be printed on three lines under each other, and packing all the codes into one string.

It's not yet perfect, though, because this string can be printed in one position only. The code 13 causes the printing to start at the extreme left-hand side, so that if we use PRINTTAB(15)A$, only the first set of blocks will be correctly TABbed. We can get around this by using other codes, as shown in Fig. 7.7. There are two versions of this program. In the first version, line 5∅ uses CHR$(1∅) to move the cursor one line down, and this is followed by five sets of CHR$(8) to shift the cursor back five spaces to the correct printing position for the next line. We need five rather than three because of the way in which the cursor moves on after printing. The second version of the program uses a FOR...NEXT loop to put in all of the CHR$(8) characters – it saves a lot of typing! This alien figure can now be printed anywhere you like on the screen by the simple command PRINTTAB(P)A$, where P is the position along the line where you want the shape to appear.

```
 5 REM ALIEN3
 10 A$=""
 20 FORN=1TO3
 30 FORX=1TO6
 40 READJ:A$=A$+CHR$(J):NEXT
 50 A$=A$+CHR$(10)+CHR$(8)+CHR$(8)+CHR$(
8)+CHR$(8)+CHR$(8)
 60 NEXT
 70 PRINT A$
 100 DATA32,27,73,80,95,80
 110 DATA32,27,73,33,95,34
 120 DATA32,27,73,62,35,77
```

(a)

```
 5 REM ALIEN3
 10 A$=""
 20 FORN=1TO3
 30 FORX=1TO6
 40 READJ:A$=A$+CHR$(J):NEXT
 50 A$=A$+CHR$(10):FORZ=1TO5:A$=A$+CHR$(
8):NEXT
 60 NEXT
 70 PRINT A$
 100 DATA32,27,73,80,95,80
 110 DATA32,27,73,33,95,34
 120 DATA32,27,73,62,35,77
```

(b)

Fig. 7.7. Using the cursor movement codes in a string – two versions.

Doubles and flashers!

Now there's quite a lot of mileage in these low resolution graphics, and we haven't finished with them by a long way. One important point to remember is that low resolution graphics take up much less memory than high resolution graphics. The ORIC is sensibly designed in this respect – even with the high resolution graphics in use you will not lose too much memory – but in a very long program, it can be wise to design for low resolution graphics rather than high resolution, particularly with the 16K machine. Just as we could obtain double height and flashing text, so we can also obtain double height and flashing graphics. The key to this is the escape codes that we use. For the 'normal' set of graphics we would use ESC I, but we have quite a few other choices.

Try, for example, clearing the screen, spacing in and down, and then typing ESC M. Any key that you press now will give you the usual graphics code – but the graphic shape flashes. The flashing rate is about twice per second, and there's probably a way of altering this by a command but I haven't found it – yet. Flashing makes a pattern very noticeable, even more so if you conceal the cursor by using CTRL Q.

Double height graphics offer another interesting variation. As with double height text, you have to start by pressing CTRL D, so do this, space along, and press ESC K. This time, when you press keys you will get all of the graphics blocks in double height – useful for monster aliens, or aliens who have taken an expanding pill! You can make these double height blocks flash by using ESC O in place of ESC K. Another interesting variation can be obtained if you *don't* use CTRL D. If you select ESC O or ESC K without having used CTRL D, you will get only half of the double-sized pattern – but this amounts to another set of graphics shapes! There's a lot to experiment with here.

PAPER and INK

The ORIC allows both text and graphics to be in colour, and if you want to experiment with these colours then you will at this stage have to use a colour TV to view the results. A black/white TV will display the colours as shades of grey, but the effect is definitely not so useful.

Let's start with PAPER and INK. As we've indicated earlier, PAPER means the background, the colour of the whole screen when no printing is present. INK means the foreground colour, the colour in which any characters or graphics will be printed on the screen. Both PAPER and INK are BASIC instruction words for the ORIC, and each must be followed by a number in the range ∅ to 7. The table in Fig. 7.8 shows what these numbers correspond to. Some of the colour names may be unfamiliar, in particular CYAN and

Number	*Colour*	*Number*	*Colour*
∅	Black	4	Blue
1	Red	5	Magenta
2	Green	6	Cyan
3	Yellow	7	White

Fig. 7.8. The PAPER and INK colours and numbers.

MAGENTA. Cyan is the colour of light that is obtained when blue and green light are mixed. Magenta is the colour of light that we get by mixing red and blue lights (yellow is a mixture of red and green lights). These colours are *not* the same as you get by mixing paints – light beams do not behave in the same way as the colours in paints.

The little program of Fig. 7.9 shows what the effects of different

```
5 REM PAPER
10 FOR N=0TO7
20 PAPER N
30 STOP
40 NEXT
```

Fig. 7.9. A program to demonstrate the PAPER (background) colours.

PAPER colours are, without any distractions due to printing. You can see, for example, that the black lettering does not show up well against some of the darker colours, and that the cursor is arranged so that its colour is always the opposite of the PAPER colour. This makes the cursor visible at all times, no matter how you change the background colour – unless you remove the cursor by using CTRL Q.

The same colour numbers are used for INK. Fig. 7.10

```
5 REM INK
10 PAPER 7
20 FOR N=0TO7
30 INK N
40 PRINT"THIS IS INK ";N
50 PRINT:WAIT500
60 NEXT
```

Fig. 7.10. A program to demonstrate the INK (foreground) colours.

demonstrates this – but note that colour applied to text is always much less satisfactory than colour applied to large areas. This is not so much a computer problem as a TV problem, and it's tied up with the way in which a TV displays colour. A colour video monitor will display very much better looking text colours than a TV receiver can.

Now that we have the facility to use PAPER and INK in different colours, we might as well flex our muscles a little. Fig. 7.11 shows how you can use INK and PAPER in a piece of text. Notice, by the way, that the PAPER and INK instructions are 'global'. That means that they affect everything that appears on the screen. Once you have selected a PAPER and INK pair of colours, they affect everything you place on the screen until you change them. That isn't necessarily what we want, so let's take a look at a much more 'free range'

```
   5 REM COLOUR TEXT
   10 CLS:PAPER5:INK3
   20 PRINT:PRINT"ORIC IS TOPS FOR COLOUR!
"
   30 GOTO30
```

Fig. 7.11. Using PAPER and INK instructions for text.

approach that lets us change colour at will, and also allows us to use colours with double height and flashing instructions.

The secret, as before, is to make use of the ESC characters. Just to show what is used, Fig. 7.12 lists the ESC characters for different

Foreground colours (for printing)		*Background colours (line colour)*	
ESC followed by ...		ESC followed by ...	
@ gives Black		P gives Black	
A ,,	Red	Q ,,	Red
B ,,	Green	R ,,	Green
C ,,	Yellow	S ,,	Yellow
D ,,	Blue	T ,,	Blue
E ,,	Magenta	U ,,	Magenta
F ,,	Cyan	V ,,	Cyan
G ,,	White	W ,,	White

Fig. 7.12. ESC characters for changing colours in a line.

background and foreground colours. These are the same colours as we use with PAPER and INK, but the difference here is that we can select these on a line-by-line or even a character-by-character basis. You can have one line with green background and blue printing, then one with red background and cyan printing, for example, no matter what PAPER and INK colours have been selected.

Take a look at the program in Fig. 7.13. The use of PAPER 1 in line 1∅ causes a red background to the whole screen, but the ESC T (obtained by using CHR$(27);"T") causes the line that is used for printing to be blue. The ESC C similarly causes the printing to be in yellow. The result is that we have three colours on the screen. Now

```
5 REM ESC COLOUR
10 CLS:PAPER1
20 PRINT:PRINTCHR$(27);
30 PRINT"T";CHR$(27)"C";
40 PRINT"ORIC-1 WINS AGAIN!"
```

Fig. 7.13. Demonstrating how the ESC characters control the colour in one line, while PAPER and INK control the rest of the screen area.

add some more lines, as in Fig. 7.14, and you will see that we can print another line which has a different background and print set of colours.

Now for the real fun. We can start combining the other tricks that

```
5 REM ESC COLOUR
10 CLS:PAPER1
20 PRINT:PRINTCHR$(27);
30 PRINT"T";CHR$(27)"C";
40 PRINT"ORIC-1 WINS AGAIN!"
50 PRINT
60 PRINTCHR$(27);"U";CHR$(27);"D";
70 PRINT"WITH COLOURS ON THE SCREEN"
```

Fig. 7.14. Adding more lines to the program to demonstrate other effects.

we can play using the ESC characters. Try Fig. 7.15, which illustrates one of the possibilities. We have set the PAPER colour in line 1Ø to give a yellow background over the main part of the screen. In line 2Ø, we have used PRINT to space down to the correct

```
 5 REM DH COLOUR
 10 CLS:PAPER 3
 20 PRINT:PRINTCHR$(4);
 30 PRINTCHR$(27);"V";CHR$(27);"D";
 40 PRINTCHR$(27);"J NEW WORLDS OF ORIC
ARE HERE!"
 50 PRINTCHR$(4)
```

Fig. 7.15. Printing text with several ESC characters combined.

position to get double height characters, by means of CHR$(4). Line 3Ø uses ESC V and ESC D to set a background colour of cyan and a foreground colour of blue for the line that is printed, and then line 4Ø prints the phrase in double height characters. We have used ESC J, with the J in the phrase, to get double height, and we could have used N to get double height combined with flashing if we wanted. Note that the escape characters take up some room on the line, so that you can't start the printing at the left-hand side of the screen.

These colours are not, of course, confined to text. We can pack a string with graphics characters and display this in colour as well, as Fig. 7.16 shows. You have to take some care here, however, because the length of string seems to affect the display. If line 1Ø spilled over by printing more) and 8 characters, then only these actual characters, ')' and '8' (not the graphics characters) could be printed. This may have been a peculiarity of the very early model of ORIC that I was using, but it's more likely that it was caused by the space that the ESC characters took up in the line. If you find these commands giving trouble, try confining the characters to one line as I have done on Fig. 7.16.

The next step – high resolution – will be dealt with in Chapter 8.

```
 5 REM PATTERNS
 10 A$=CHR$(27)+"O)8)8)8)8)8)8)8)8"
 25 CLS
 30 PAPER5
 35 PRINT
 40 PRINTCHR$(4);
 50 PRINTCHR$(27);"T";CHR$(27);"B";
 70 PRINTTAB(12)A$:PRINT
 80 PRINT:PRINT
 100 PRINTCHR$(27);"S";CHR$(27);"A";
 110 PRINTCHR$(27);"J SPECTACULAR DISPLA
YS ARE POSSIBLE"
 120 PRINTCHR$(4);CHR$(17)
 130 END
```

Fig. 7.16. Graphics characters printed in colour.

Chapter Eight
Graphics (2)

We can make quite a lot of use of the low resolution graphics, particularly because they allow us to use a lot of the memory of the ORIC for programming. For really spectacular graphics, though, we need to use high resolution, taking about 7100 bytes of memory. On the 48K or 32K ORIC, this is a negligible amount, but on the 16K ORIC, you could find that the combination of high resolution graphics and a long program could leave you with very little memory left. The appearance of good high resolution graphics programs, however, can be very spectacular. This is further enhanced if animation is added, so that the graphics shapes are moved around.

In this chapter, we're going to look at two types of high resolution graphics that the ORIC permits. These are *pattern drawing*, and *user-defined characters*. Pattern drawing means the ability to create lines on the screen where you want them, and in any of the range of colours that the ORIC can use. User-defined characters are shapes, which could be different letters (Greek, perhaps?), or 'space invader' shapes, and these can be moved around the screen by relatively simple instructions. We'll deal first with the line drawing instructions.

Straight line drawing

To draw a line, we need to specify a starting point. The ORIC has two methods of specifying the start of a line – CURSET and CURMOV. Both of these use *position co-ordinates*, numbers which determine a position on the screen. The co-ordinates for the high-resolution display are illustrated in Fig. 8.1. There are 240 points, numbered Ø to 239, across the screen, with position Ø at the left-hand side of the screen. These numbers, Ø to 239, are the X co-ordinate numbers. The Y co-ordinate numbers range from Ø to 199, down the

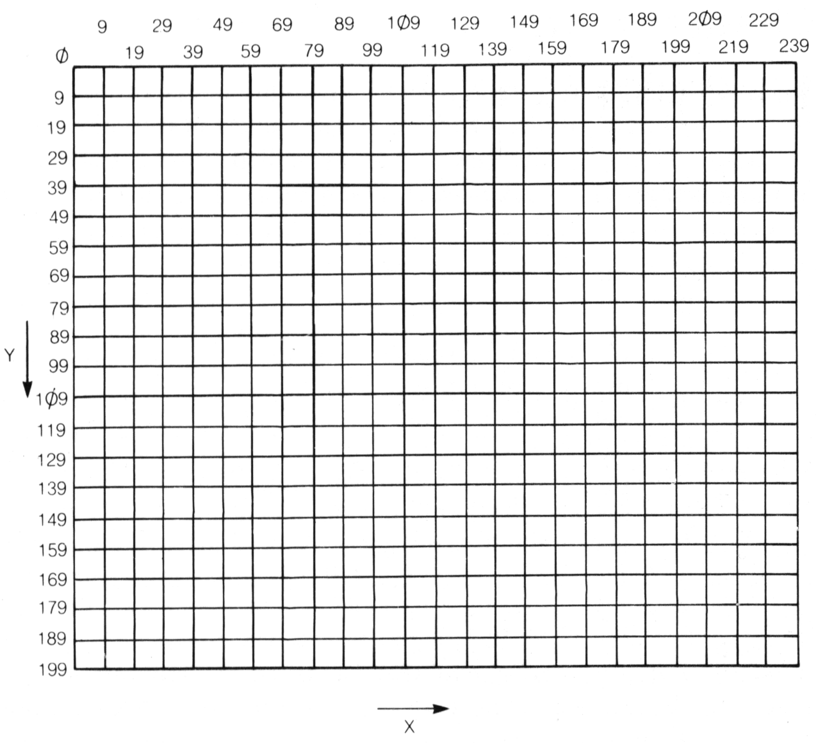


Fig. 8.1. The co-ordinate diagram for high resolution graphics.

screen, starting at the top of the screen and numbering downwards. The position X = ∅, Y = ∅ is the top left-hand corner of the screen. The position X = 239, Y = 199 is the bottom right-hand corner. You must be careful in any high resolution graphics program that you do not attempt to use numbers outside these ranges.

These co-ordinate numbers are very important because all of the high resolution drawing instructions use them. Some instructions are *absolute*, however, and some are *relative*. An absolute instruction means one which uses the X and Y co-ordinates of the screen positions – using the same pair of numbers will always get you to the same place. A relative instruction uses its co-ordinate numbers differently – it adds them to the co-ordinate numbers that the previous instruction used! Using an instruction with relative co-ordinates is rather like saying 'go three steps forward and two left'. Unless you know where you were before, it's not easy to work out where you will end up. Relative movement is useful, however, if you are tracing a shape which may have to be drawn at different positions on the screen, as we'll see later.

Down to work. CURSET is a cursor position instruction which has to be followed by three numbers, all separated by commas. The first of these numbers is the X co-ordinate number, and the second is the Y co-ordinate number, keeping to the ranges of Ø to 239 for X and Ø to 199 to Y. The last number is called the *FB number* – meaning foreground/background. Fig. 8.2 lists these numbers, Ø to 3, which control the colour of the cursor on the high resolution screen. Unlike the cursor that we use for text, the high resolution cursor is a dot that does not flash, and which can be kept invisible by using a FB number of Ø. The FB codes allow you to convert the cursor colour to background colour (making it invisible), to foreground colour, to swap foreground and background colours, or to leave things as they are. You must, however, put *one* of these numbers into the instruction.

FB Number	Effect
Ø	Use background colour – this will show only if the cursor is over a piece of foreground.
1	Use foreground – this will show if the cursor is over background.
2	Invert colours – draw in the colour that is the *opposite* of the one used for the previous instruction.
3	Do nothing – do not place anything on the screen.

Fig. 8.2. The FB numbers Ø to 3.

CURMOV is the relative-move instruction. The X and Y numbers that follow CURMOV are added to the X and Y numbers that were used by the previous instruction. For example, suppose we set the cursor to the centre of the screen by using CURSET 119,99,1. If we now use the instruction CURMOV2Ø,1Ø,1, then the cursor moves to 139,1Ø9, which is 2Ø more on X and 1Ø more on Y. We can move in either direction, because the numbers that follow CURMOV can be negative or positive. With the cursor at the centre of the screen, for example, CURMOV–2Ø,–1Ø,1 will place the cursor at the position 99,89, and with the cursor dot visible because of the FB number of 1.

High resolve

Now we can start to look at these instructions in action. When we use ORIC normally for programming, we make use of the TEXT screen. This uses a small amount of memory, and is suited for characters and for low resolution graphics. To use the high resolution graphics, we need to use more memory to store the information that we see on the screen. This is done by using the instruction or command HIRES. When you type HIRES as a direct command, and press RETURN, the text on the screen clears and the screen goes dark unless you specified another PAPER colour. Unlike most computers, ORIC allows you to keep an eye on what you are doing. The three lines under the main screen area are what is called the *text window*. We can type instructions and list programs using this area, so that it's particularly easy to see what the effects of commands are. We'll use this to take a tour around the screen.

Type HIRES and press RETURN. Now type CURSET 79,149,ϕ to put the cursor in position on the screen. You still can't see the cursor because we have used the FB number ϕ. Now we're going to trace the pattern shown in Fig. 8.3. Tracing is best done by placing a piece of tracing paper over the high resolution grid in Fig. 8.1, and sketching in pencil. Decide on a starting position, a corner, and note its co-ordinates. Then pencil in the co-ordinates of all the other points were the straight lines change direction. We've shown this in Fig. 8.3 – the alternative is to write the starting co-ordinates and then to use relative numbers. I always prefer to work with absolute co-ordinates, because I can then be sure of getting back to where I started from in a pattern like this.

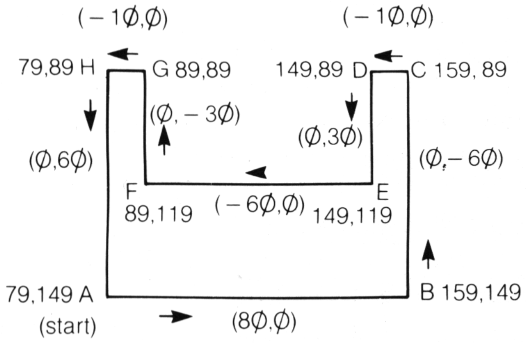


Fig. 8.3. Tracing a pattern so that it can be programmed easily.

Once the tracing has been made, we can work out the relative co-ordinate numbers by subtraction. For the first line, for example, we subtract the starting co-ordinates of 79,149 from the next position co-ordinates of 159,149. This gives us 8ϕ,ϕ as the relative

```
DRAW 8ø,ø,1
DRAW ø,-6ø,1
DRAW -1ø,ø,1
DRAW ø,3ø,1
DRAW -4ø,ø,1
DRAW ø,-3ø,1
DRAW -1ø,ø,1
DRAW ø,6ø,1
```

Fig. 8.4. DRAW instructions that can be entered as direct commands.

co-ordinate numbers. We have subtracted the old X from the new X, and the old Y from the new Y. The next set is calculated from points B to C in the same way, and the same applies to all the others.

Now we're ready for a new instruction – DRAW. Fig. 8.4 shows the DRAW instructions that can be entered one by one. You don't need to make a program out of this, just type each instruction and press RETURN. As you do so, you can watch the results appear on the screen – one of the considerable advantages of ORIC-1

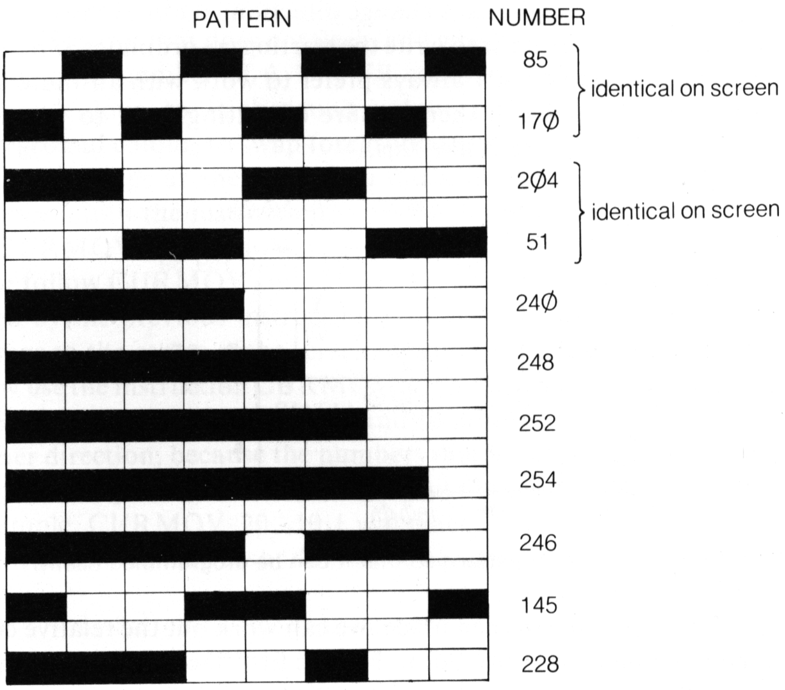


Fig. 8.5. Some line patterns obtained by using the PATTERN instruction.

computing. You will find that if you enter an instruction that would cause the cursor to disappear from the screen, you will get an error message.

The lines that you draw on the screen in this way may be solid, dotted or dashed. A line is normally solid, but other types of lines can be obtained by using the PATTERN instruction. Fig. 8.5 shows some of the patterns that can be obtained by using different numbers following PATTERN. Appendix D shows how these pattern numbers are obtained, and how you can create the pattern of your choice.

High resolution programming

We can make use of direct commands to experiment with high resolution graphics drawing, but we normally do so with the aim of making a program. Let's take an example and analyse it. We start by tracing the shape that we want to draw, as in Fig. 8.6. This is

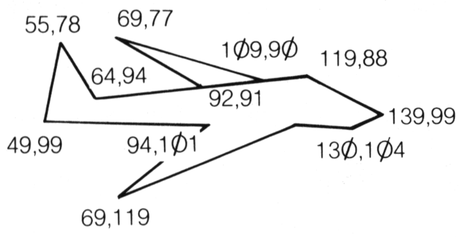


Fig. 8.6. A shape that we want to produce on the screen.

considerably more complicated than the shapes that we have used so far, so we're flexing our muscles a bit on this one. The drawing is shown with its absolute co-ordinates, and we shall have to calculate the *relative* co-ordinate numbers for each DRAW instruction.

The program is illustrated in Fig. 8.7. It has been written as a main section, lines 5 to 4∅, and a subroutine with the subroutine containing the DRAW instructions. We'll see later why a subroutine has been used as we develop this program – it allows us to animate the drawing. Each DRAW uses two co-ordinates, and the FB of 1 which draws the line in the foreground (INK) colour. The only part of this program that might confuse you is the CURMOV in line 1∅2∅. The second line that is drawn in line 1∅1∅ goes to the rear of the wing, at point 92,91, and I have used CURMOV to shift the cursor back to 1∅9, 9∅ so as to draw the wing itself. I could have used another

```
5 REM AERO1
10 HIRES
20 CURSET139,99,0
30 GOSUB1000
40 END
1000 DRAW-20,-11,1
1010 DRAW-27,3,1
1020 CURMOV17,-1,1
1030 DRAW-40,-13,1
1040 DRAW23,14,1
1050 DRAW-28,3,1
1060 DRAW-9,-16,1
1070 DRAW-6,21,1
1080 DRAW45,2,1
1090 DRAW-25,18,1
1100 DRAW47,-17,1
1110 DRAW14,2,1
1120 DRAW9,-5,1
1130 RETURN
```

Fig. 8.7. A program for producing the shape on the screen.

DRAW instruction, but it looked like a good place to demonstrate this use of CURMOV!

When the program runs, you will see the shape appear in a reasonably short time. You can wipe the drawing by typing TEXT, which returns the ORIC to the text screen. This does not completely remove the drawing from memory, because pressing ESC DEL in sequence will restore it (or part of it, on the early model) though you

```
5 REM AERO2
10 HIRES
20 CURSET139,99,0
30 N=1
40 GOSUB1000
50 END
1000 DRAW-20,-11,N
1010 DRAW-27,3,N
1020 CURMOV17,-1,N
1030 DRAW-40,-13,N
1040 DRAW23,14,N
1050 DRAW-28,3,N
1060 DRAW-9,-16,N
1070 DRAW-6,21,N
1080 DRAW45,2,N
1090 DRAW-25,18,N
1100 DRAW47,-17,N
1110 DRAW14,2,N
1120 DRAW9,-5,N
1130 RETURN
```

Fig. 8.8. Using a number variable as the FB number allows drawings to be made and then 'wiped'.

will need to use the restore button under the ORIC to recover from this afterwards.

These images can be much more easily drawn and removed if the program is written with a number variable in place of the 1 that we have used as the FB number in each DRAW. Fig. 8.8 shows this done, with the value of N set in line 3∅ before the subroutine is called. Now we can see the usefulness of a subroutine. Fig. 8.9 shows

```
5 REM AERO2
10 HIRES
20 CURSET139,99,0
30 N=1
40 GOSUB1000
50 WAIT1000
60 N=0
70 GOSUB1000
80 END
1000 DRAW-20,-11,N
1010 DRAW-27,3,N
1020 CURMOV17,-1,N
1030 DRAW-40,-13,N
1040 DRAW23,14,N
1050 DRAW-28,3,N
1060 DRAW-9,-16,N
1070 DRAW-6,21,N
1080 DRAW45,2,N
1090 DRAW-25,18,N
1100 DRAW47,-17,N
1110 DRAW14,2,N
1120 DRAW9,-5,N
1130 RETURN
```

Fig. 8.9. A 'draw and wipe' program, using a variable as the FB number.

the routine extended so that the shape is drawn, and then removed. The removal is done simply by repeating the drawing steps with N = ∅ instead of N = 1. This is possible only if the cursor starts at the same position – exactly – for removal as it had when it traced out the lines. We have to be careful, then, when we use this method that we always return the cursor to the correct position. If a pattern uses a different finishing point, then a CURMOV or CURSET will be needed before the subroutine is called to erase the drawing.

Fig. 8.10 shows an attempt to animate this drawing. The normal method of animation consists of creating the drawing, waiting for a short time, then erasing the image, waiting, then creating another identical drawing in a slightly different position. If this can be done fast enough and smoothly enough, an illusion of movement can be created. The program in Fig. 8.10 runs too slowly, however, and the successive drawing and erasing looks too obvious in a drawing of

```
5 REM AERO2
10 HIRES
20 FORJ=120TO230
30 CURSETJ,99,0:N=1
40 GOSUB1000
50 N=0:GOSUB1000
60 NEXT:N=1:GOSUB1000
70 END
1000 DRAW-20,-11,N
1010 DRAW-27,3,N
1020 CURMOV17,-1,N
1030 DRAW-40,-13,N
1040 DRAW23,14,N
1050 DRAW-28,3,N
1060 DRAW-9,-16,N
1070 DRAW-6,21,N
1080 DRAW45,2,N
1090 DRAW-25,18,N
1100 DRAW47,-17,N
1110 DRAW14,2,N
1120 DRAW9,-5,N
1130 RETURN
```

Fig. 8.10. A first attempt at animation.

this size. A slight improvement results if we do the drawing at the next position before we erase the first one, and this technique is illustrated in Fig. 8.11. A further improvement is obtained by using a larger step size (Fig. 8.12) and the combination of a larger step and a

```
5 REM AERO2
10 HIRES
20 FORJ=120TO230STEP2
30 CURSETJ,99,0:N=1:GOSUB1000
40 CURSETJ+2,99,0:N=1:GOSUB1000
50 CURSETJ,99,0:N=0:GOSUB1000
60 NEXT
70 END
1000 DRAW-20,-11,N
1010 DRAW-27,3,N
1020 CURMOV17,-1,N
1030 DRAW-40,-13,N
1040 DRAW23,14,N
1050 DRAW-28,3,N
1060 DRAW-9,-16,N
1070 DRAW-6,21,N
1080 DRAW45,2,N
1090 DRAW-25,18,N
1100 DRAW47,-17,N
1110 DRAW14,2,N
1120 DRAW9,-5,N
1130 RETURN
```

Fig. 8.11. Improving the appearance of animation by carrying out the next drawing before erasing the first of any pair of drawings.

```
5 REM AERO2
10 HIRES
20 FORJ=120TO230
30 CURSETJ,99,0:N=1:GOSUB1000
40 CURSETJ+1,99,0:N=1:GOSUB1000
50 CURSETJ,99,0:N=0:GOSUB1000
60 NEXT
70 END
1000 DRAW-20,-11,N
1010 DRAW-27,3,N
1020 CURMOV17,-1,N
1030 DRAW-40,-13,N
1040 DRAW23,14,N
1050 DRAW-28,3,N
1060 DRAW-9,-16,N
1070 DRAW-6,21,N
1080 DRAW45,2,N
1090 DRAW-25,18,N
1100 DRAW47,-17,N
1110 DRAW14,2,N
1120 DRAW9,-5,N
1130 RETURN
```

Fig. 8.12. Improving animation by using a larger step.

pause, using a FOR...NEXT loop, is better still (Fig. 8.13).

None of these animations is truly convincing, however, because the BASIC programming language of the DRAW instruction is just too slow for really good animation. To animate drawings of this size, we need to use a simpler but faster type of language for programming. This could be machine code or, better still, a programming language like FORTH that is more suitable for fast animation. The 48K ORIC is now being supplied with FORTH as well as with BASIC.

Circling around

Drawing with straight lines alone is not enough for many of the patterns that we want to draw, and the ORIC permits the use of a

```
20 FOR J = 120 TO 230 STEP 5
35 FOR X = 1 TO 20:NEXT
40 CURSET J+5,99,0:N=1:GOSUB 1000
45 FOR N = 1 TO 20:NEXT
55 FOR N = 1 TO 20:NEXT
```

Fig. 8.13. Lines which add a larger step and a pause.

CIRCLE instruction. This draws a circle whose centre is at the cursor position, and all we have to specify is the radius and the FB code.

It was probably a peculiarity of my early model ORIC, but I found that the minimum radius for CIRCLE was 8. Any attempt to use a smaller figure caused a solid circle to be drawn, and the computer then either locked up or drew random patterns. As it happens, a radius figure of 8 gives just about the smallest circle that still looks reasonably circular.

The program of Fig. 8.14 shows a car shape that has been created using the DRAW, CURSET, CURMOV and CIRCLE instructions. Each portion has been drawn using a different subroutine, and a new

```
5 REM CAR
10 HIRES
20 CURSET159,107,0:N=1
30 GOSUB1000:REM CAR
40 GOSUB2000:REM WHEELS
50 GOSUB3000:REM TEXT
60 END
1000 DRAW0,-3,N
1010 DRAW-2,-2,N
1020 DRAW-1,-4,N
1030 DRAW-9,0,N
1040 DRAW-22,-5,N
1050 DRAW-10,-8,N
1060 DRAW-17,3,N
1070 DRAW-19,4,N
1080 DRAW-12,7,N
1090 DRAW1,8,N
1100 DRAW6,1,N
1110 CURMOV13,0,0
1120 DRAW55,-1,N
1130 CURMOV13,0,0
1140 DRAW4,0,N
1150 RETURN
2000 CURSET80,108,0
2010 CIRCLE8,1
2020 CURSET149,107,0
2030 CIRCLE8,1
2040 RETURN
3000 CURSET90,96,0
3010 TX$="Turbo"
3020 FORX=1TO LEN(TX$)
3030 CHAR ASC(MID$(TX$,X,1)),0,1
3040 CURMOV6,0,0
3050 NEXT
3060 RETURN
```

Fig. 8.14. Using the CIRCLE instruction with DRAW. On the early model of ORIC, a CIRCLE radius number of less than 8 caused problems.

trick has been incorporated – adding text to a drawing. Many computers do not permit text to be placed directly on the high resolution screen, so the ability to do this is a useful and unusual feature of the ORIC. The instruction that is used is CHAR, which has to be followed by three numbers. The first number is the ASCII code for the letter, the second number is Ø if we use the normal text characters, and the third is the usual FB number which will be 1 if we are using foreground colour. CHAR prints one character only, and does *not* cause the cursor to move, so that unless we move the cursor between two CHAR instructions, the letters will appear one on top of the other – a feature which can sometimes be useful in order to create special effects.

The loop in lines 3Ø2Ø to 3Ø5Ø extracts characters one by one from the string TX$, finds their ASCII codes, and then places the ASCII codes one by one into a CHAR instruction. The CURMOV in line 3Ø4Ø is used to space the letters along the side of the car. The normal spacing between letters is 1Ø, but we have used 6 here to avoid making the word too long in relation to the size of the car shape.

As a final touch, we can make the wheels appear to turn! Fig. 8.15 shows how this is done, drawing the wheels with one pattern, erasing it, and then using a 'complementary' pattern. The complementary pattern is one which has dots where the first one has spaces, and spaces where the first one has dots. The effect is not perfect, because of the small size of the wheels, but it shows how the action is carried out.

Making a point

So far, the instructions that we have looked at will move the cursor around and draw lines from a cursor position. There is another instruction, POINT, which is used to detect whether the cursor is touching foreground or background colour. The POINT instruction needs the X and Y values, enclosed in brackets, and it gives one of two numbers. The number will be Ø if the point X,Y is at background colour, and −1 if the point X,Y is at foreground colour.

POINT is used as a way of detecting if the cursor position is over some part of the screen which is at a different colour. If, for example, you are moving a graphics block across the screen by using instructions like:

```
5 REM WHEELS!
10 HIRES
20 CURSET159,107,0:N=1
30 GOSUB1000:REM CAR
40 GOSUB3000:REM TEXT
45 PATTERN51:Z=1:GOSUB2000
46 Z=0:GOSUB2000
50 PATTERN204:Z=1:GOSUB2000
51 Z=0:GOSUB2000
55 GOTO45
60 END
1000 DRAW0,-3,N
1010 DRAW-2,-2,N
1020 DRAW-1,-4,N
1030 DRAW-9,0,N
1040 DRAW-22,-5,N
1050 DRAW-10,-8,N
1060 DRAW-17,3,N
1070 DRAW-19,4,N
1080 DRAW-12,7,N
1090 DRAW1,8,N
1100 DRAW6,1,N
1110 CURMOV13,0,0
1120 DRAW55,-1,N
1130 CURMOV13,0,0
1140 DRAW4,0,N
1150 RETURN
2000 CURSET80,108,0
2010 CIRCLE8,Z
2020 CURSET149,107,0
2030 CIRCLE8,Z
2040 RETURN
3000 CURSET90,96,0
3010 TX$="Turbo"
3020 FORX=1TO LEN(TX$)
3030 CHAR ASC(MID$(TX$,X,1)),0,1
3040 CURMOV6,0,0
3050 NEXT
3060 RETURN
```

Fig. 8.15. Making the car wheels appear to move!

```
CURMOV X,∅,∅
```

in a loop, then you may want some action to occur if the cursor hits a band of colour that represents earth, an enemy submarine, an alien, or whatever you like. We can program this by using POINT. Suppose we want to reverse the movement of the cursor if it hits a piece of material that is of foreground colour. In this case, POINT will change from ∅ to −1 when the cursor X and Y values take it anywhere within the material. We can then reverse the direction of movement by using, for example, IF POINT(X,Y) = −1 THEN X = −X

Fig. 8.16 illustrates this type of action in a program that bounces a dot between lines. Another possibility is to make the detecting action of POINT cause a piece of material to be obliterated (by drawing it again in the background colour) and a sound to be generated. Sound generation is dealt with in Chapter 9, but it's not giving any secrets away to say that putting a line like:

IF POINT(X,Y) = −1 THEN EXPLODE

can produce some very interesting effects!

```
 5 REM POINT
 10 HIRES:X=1
 15 Z=1
 20 CURSET0,0,0
 25 PRINTCHR$(17)
 30 DRAW0,199,1
 40 CURSET239,0,0
 50 DRAW0,199,1
 60 CURSETX,100,1
 80 CURSETX,100,0
 90 X=X+Z
 100 IF POINT(X,100)=-1THENZ=-Z:PING:X=X
+Z
 110 GOTO60
```

Fig. 8.16. Using POINT to reverse the direction of the cursor.

Chapter Nine
New Characters and Sounds

The drawings that are made using the DRAW and CIRCLE instructions are better suited to non-moving graphics. The ORIC therefore allows you to create shapes which are more easily manipulated, using what is called *user-defined characters.* The normal characters of the ORIC are the letters, digits and punctuation marks, and the alternate set consists of the low resolution graphics characters, but we can also create and store our own shapes. Each created character is small, with a maximum size about the same as the cursor block, but we can, of course, combine these characters into a string just as we do with normal characters. We can then place our special characters on to the high resolution screen using the CHAR instruction with the ASCII codes, or we can print the character on to the TEXT screen using the PRINT instruction. While we illustrate this process, we'll also look at some of the pre-programmed sound effects of the ORIC.

Before we make a start to this work, however, we need to have some idea about what we are doing. The key to it all is the memory of the ORIC. The ORIC, like all computers, has two lots of memory. One set, called the ROM is fixed and unchanging. The ROM, meaning Read-Only Memory, is always in use controlling the actions of the computer. The rest of the memory, called RAM (Random Access Memory) can be altered as we please, and loses anything that was stored in it when the power is switched off. It is the RAM that we use to store our programs and to store what appears on the screen.

The RAM is also used to store the data numbers that are codes for character shapes. Each unit of memory is called a *byte*, and one byte can store number codes that range in value from ∅ up to 255. We use these codes in various ways – you have already met the ASCII codes which are used to specify characters in programs. When a character is to be displayed on the screen, though, a single number code is not enough. What we need for this purpose is a set of number codes that will build up the correct pattern on the screen. Fig. 9.1, for example, shows a matrix of 64 squares, with some of the squares shaded so as

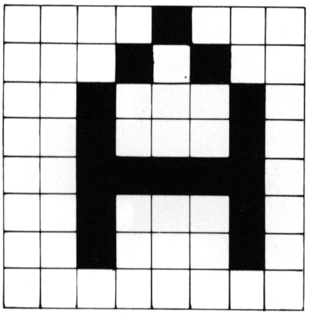


Fig. 9.1. The 8 × 8 matrix which is used to plan a character on to the screen.

to form an A shape. A pattern like this with eight blocks across can be coded using one byte of memory for each row. To do this, we allocate a number for each unit in a row, as shown in Fig. 9.2. The left-hand side unit is allocated the number 128, the right-hand side

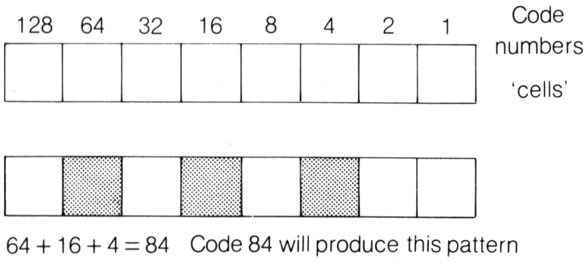


Fig. 9.2. Numbering the blocks in a row of the 8 × 8 matrix.

unit with 1, and each number is half of the previous one, as we read from left to right. This is the famous *binary number scale* that you may have read about, but we don't have to worry about what it's called. The point is that we can deal with a row of eight blocks like this by adding up the numbers for the shaded blocks. Fig. 9.2 also shows an example of a line of eight blocks of which three are shaded. The shaded blocks are the ones which carry the coding of 64,16 and 4, so that the code for the whole set is 64 + 16 + 4 = 84. To make up a complete character, we need a set of eight rows of blocks like this, as illustrated in Fig. 9.3. Each of the rows can be represented by a number, and each of the numbers can be placed in the memory of the computer so as to define the complete shape.

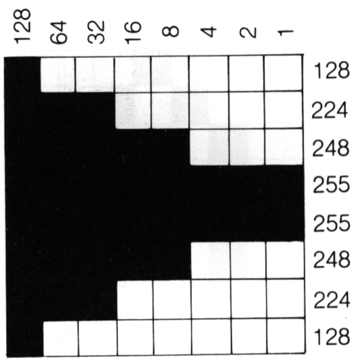


Fig. 9.3. Obtaining the codes for a complete character.

Now there is a section of the RAM memory which is reserved for storing these numbers. There are 128 normal characters, and since each character needs eight bytes, the normal set of characters uses $128 \times 8 = 1024$ bytes (which is 1K of memory). We distinguish where each byte is placed in the memory by using numbers again, this time called *address numbers*. On the 48K ORIC that I am using, the first byte of the normal character set starts at address 46080. For the 16K ORIC, the starting address would be 13312. There is also an alternate character set which is used to store the low resolution graphics characters. This starts at 47104 on the 48K ORIC and at 14336 on the 16K machine. The point about these address numbers is that we can use them to find out what is stored at each memory address, and that we can also use them to change what is stored. Take a look at Fig. 9.4, which is a program designed to reveal the

```
5 REM FIND CHAR
10 X=46080
15 Z=8*125
20 FORN=0TO7
25 PRINTPEEK(X+Z+N)
30 NEXT
40 REM CHAR.125 IS CURLY BRACKET
```

Fig. 9.4. A program designed to display the contents of memory. The PEEK instruction finds what is stored at the address number that follows the instruction (in brackets).

contents of eight bytes of memory. These are the addresses where the data for the curly-bracket character which has the ASCII code of 125 is stored. When we analyse these numbers into units of

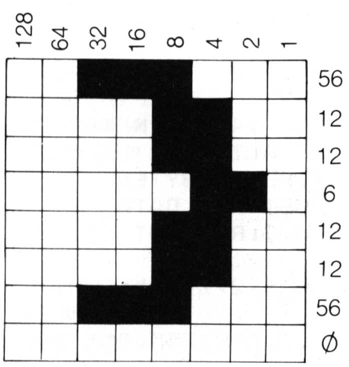


Fig. 9.5. Shading in blocks so as to find what shape is stored in a set of eight memory addresses.

128,64,32,16,8,4,2 and 1, we can shade in blocks on a 8 × 8 diagram so as to see the shape, as in Fig. 9.5.

We can now replace these bytes by others. Suppose we use the shape of Fig. 9.3. We can place the list of numbers that forms this shape into the same addresses in the memory by using the program of Fig. 9.6. This uses the instruction POKE which has to be followed

```
5 REM WEDGE
10 CH=46080:REM 13312 IN 16K ORIC
20 AD=8*125:REM 8 BYTES FOR ASCII 125
30 FOR N=0TO7:REM 8 BYTES
40 READJ:REM GET NEW DATA
50 POKECH+AD+N,J:REM PUT IT IN
60 NEXT
70 PRINT:PRINTCHR$(125)
100 END
500 DATA128,224,248,255,255,248,224,128
```

Fig. 9.6. A program which changes the numbers stored in memory, using the POKE instruction.

by two numbers. The first of these numbers is the address into which the data has to be placed. The second number is the data byte – a number between Ø and 225. The program therefore places the correct byte numbers for our shape into these memory addresses in place of the bytes for the curly-bracket character. This means that our own character will now replace the curly-bracket character in the memory, so that if we use the key that prints the curly-bracket character we will get the new character instead. We can also use within a program the instruction PRINT"}" or PRINTCHR$(125) or, on the high resolution screen, CHAR125 to put this character in place.

We can also use addresses in the alternate character set, characters with ASCII codes from 128 onwards. These have the addresses that are higher than the normal set, and Fig. 9.7 shows an example of one of these alternate characters being replaced. This type of character is

```
5 REM ALT WEDGE
10 CH=47104:REM 14336 IN 16K ORIC
20 AD=8*33:REM ALT.!
30 FOR N=0TO7:REM 8 BYTES
40 READJ:REM GET NEW DATA
50 POKECH+AD+N,J:REM PUT IT IN
60 NEXT
70 PRINT" ";CHR$(27);"I";CHR$(33)
100 END
500 DATA128,224,248,255,255,248,224,128
```

Fig. 9.7. Replacing one of the alternate character set.

most easily printed simply by using its ASCII code. Note, incidentally, that when you press the reset button under the body of the ORIC, all of the characters go back to their normal definitions again. To obtain your new character, you will have to re-run the program that poked the numbers into the memory.

Using user-designed characters

Now that we have the ability to create characters to our own designs, what do we do with them? One application that springs to mind is our Greek days-of-the-week program – we could now design a set of Greek letters to print out the correct Greek names instead of using pronunciations! In general, we can use the user-defined characters on either type of screen, TEXT or HIRES, and because of the small size of the characters, they can be animated more convincingly than the larger shapes that are created by the DRAW and CIRCLE instructions. The animation is best done by using the high resolution screen, because this allows us to use smaller steps for movement.

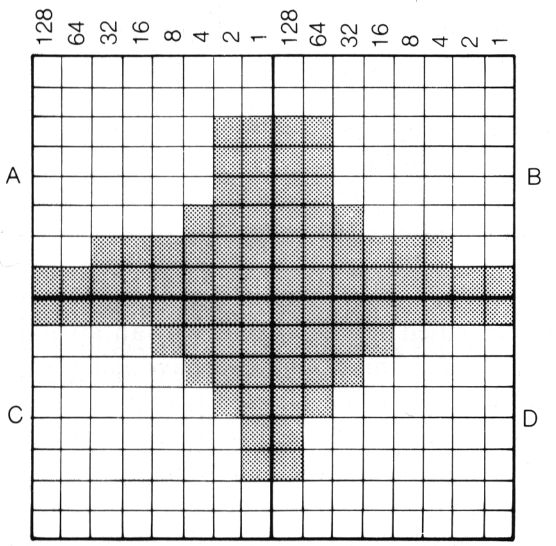
Fig. 9.8 shows an example of animation of the character that we have created. This is done by printing the character, using CHAR on the high resolution screen at the cursor position, deleting the character by printing it in background colour, then moving the cursor and repeating the process. Since the cursor can be moved in any direction, the character can also be moved in any direction, and a single PRINT or CHAR instruction is very much faster to carry out than a set of DRAW instructions. In Fig. 9.8, we can move the character faster by altering the amount of cursor movement and the range of X. Try CURMOV4,∅,∅ and FOR X = 1 TO 58 in the program.

```
5 REM FAST WEDGE
10 CH=47104:REM 14336 IN 16K ORIC
20 AD=8*33:REM ALT.!
30 FOR N=0TO7:REM 8 BYTES
40 READJ:REM GET NEW DATA
50 POKECH+AD+N,J:REM PUT IT IN
60 NEXT
70 HIRES
80 CURSET0,100,0
90 FORX=1TO58
100 CHAR33,1,1
110 FORN=1TO20:NEXT
120 CHAR33,1,0
130 CURMOV4,0,0
140 NEXT
200 END
500 DATA128,224,248,255,255,248,224,128
```

Fig. 9.8. Animating a 'user-designed' character.

We can, of course, design sets of characters that fit together so as to create a large object. When this is done, it takes slightly longer to place the character on the screen because we have to print a string of characters. When we use the high resolution screen, we can use CHAR and CURMOV to place different characters in different places, and overlap them or space them as we want.

Fig. 9.9 shows a flying saucer shape created from four characters,



Ø	Ø
Ø	Ø
3	192
3	192
3	192
7	224
63	236
225	225
225	225
15	24Ø
7	224
3	192
1	128
1	128

Fig. 9.9. A 'flying-saucer' shape which will require four character blocks.

with the four sections marked as A,B,C, and D. We shall use ASCII codes 6Ø,62,123 and 125 respectively for these four sections, using the methods that we have illustrated before, and printing them on the text screen. The program which creates the shapes is illustrated in Fig. 9.10. This has been printed showing the original characters in the PRINT statements but, when the program has run, these keys will give the new shapes. The next step is to carry out some animation of this shape. It seems reasonable to expect it to move

```
5 REM SAUCER
10 CH=46080:REM 13312 IN 16K ORIC
20 FOR X=1TO4:REM 4 CHARACTERS
30 READ AS:AD=8*AS:REM AS IS ASCII
40 FOR N=0TO7:READ J
50 POKE CH+AD+N,J:NEXT
60 NEXT
70 PRINT:PRINT"  ";"<>"
80 PRINT"  ";"{}"
100 END
500 DATA60,0,0,3,3,3,7,63,255
510 DATA62,0,0,192,192,192,224,252,255
520 DATA123,255,15,7,3,1,1,0,0
530 DATA125,255,240,224,192,128,128,0,0
```

Fig. 9.10. Placing the saucer shape on the screen.

down and across the screen, so we've used a formula for moving the cursor which changes the Y amount in a way that is determined by the value of X. The log of a number (this is the 'natural' logarithm, not the base ten log if you are mathematically minded) does not increase much as the number increases, so by using INT(LOG(X) *32), we get values for Y which change rapidly at first, and then level out. The result is a flashing saucer (Fig. 9.11) which rapidly descends and then coasts along, more visible, looking for the sort of human beings who might believe in it.

Now for something rather more spectacular, using the program in Fig. 9.12. This uses essentially the same program, but with three extra lines. Line 71 places the cursor at the 'home' position of ∅,∅, and then carries out a FILL instruction. The FILL instruction has the effect of filling a specified number of lines with an 'attribute' (like colour, flashing, etc.), and in this example we have picked the colour green whose code number is 2 when it is used as a foreground colour. Since the FILL instruction uses 19∅ as its line-fill number, the first number after FILL, this means that anything below the home position will appear in green when it is printed on the screen.

The next line shifts the cursor to X = 1∅∅, still on the top line, and uses a FILL instruction with colour 4, which is blue. Now this also affects 19∅ lines, but only from the X = 1∅∅ position onwards. The result of these two FILL instructions, therefore, will be to make

```
5 REM SAUCERATTACK!
10 CH=46080:REM 13312 IN 16K ORIC
20 FOR X=1TO4:REM 4 CHARACTERS
30 READ AS:AD=8*AS:REM AS IS ASCII
40 FOR N=0TO7:READ J
50 POKE CH+AD+N,J:NEXT
60 NEXT
70 HIRES
80 CURSET1,1,0
90 FORX=1TO230STEP5
100 K=1:GOSUB1000
120 K=0:GOSUB1000
130 CURSETX,INT(LOG(X)*32),0
140 NEXT
300 END
500 DATA60,0,0,3,3,3,7,63,255
510 DATA62,0,0,192,192,192,224,252,255
520 DATA123,255,15,7,3,1,1,0,0
530 DATA125,255,240,224,192,128,128,0,0
1000 CHAR60,0,K
1010 CURMOV10,0,0
1020 CHAR62,0,K
1030 CURMOV-10,8,0
1040 CHAR123,0,K
1050 CURMOV10,0,0
1060 CHAR125,0,K
1070 CURMOV-10,-8,0
1080 RETURN
```

Fig. 9.11. Animating the saucer shape.

positions X = ∅ to X = 1∅∅ give green printing, and from X = 1∅∅ onwards print in blue. By the third FILL instruction we cause X = 2∅∅ onwards to have a foreground colour of cyan.

The effect now is that the object changes colour as it moves across the screen. The middle figure in FILL,1 in our example, decides how many blank spaces are placed between the two sets of filled lines – it's really a 'not-fill' number (on my ORIC anyhow!). If you replace line 72 with CURSET1∅∅,∅,∅ : FILL 19∅,1∅,4 you will see your saucer disappear into a black hole before it reappears in blue again!

Fig. 9.13 shows the colour numbers for the FILL instructions – note that the colours used for backgrounds have numbers ranging from 16 to 23. Other numbers can be used with FILL, and the scope for experiment with these instructions alone should keep you active for quite a few weeks.

Sound and fury

The capabilities of the ORIC don't stop at its excellent graphics –

```
5 REM SAUCERATTACK!
10 CH=46080:REM 13312 IN 16K ORIC
20 FOR X=1TO4:REM 4 CHARACTERS
30 READ AS:AD=8*AS:REM AS IS ASCII
40 FOR N=0TO7:READ J
50 POKE CH+AD+N,J:NEXT
60 NEXT
70 HIRES
71 CURSET0,0,0:FILL190,1,2
72 CURSET100,0,0:FILL190,1,4
73 CURSET200,0,0:FILL190,1,6
80 CURSET1,1,0
90 FORX=1TO230STEP5
100 K=1:GOSUB1000
120 K=0:GOSUB1000
130 CURSETX,INT(LOG(X)*32),0
140 NEXT
300 END
500 DATA60,0,0,3,3,3,7,63,255
510 DATA62,0,0,192,192,192,224,252,255
520 DATA123,255,15,7,3,1,1,0,0
530 DATA125,255,240,224,192,128,128,0,0
1000 CHAR60,0,K
1010 CURMOV10,0,0
1020 CHAR62,0,K
1030 CURMOV-10,8,0
1040 CHAR123,0,K
1050 CURMOV10,0,0
1060 CHAR125,0,K
1070 CURMOV-10,-8,0
1080 RETURN
```

Fig. 9.12. Using FILL to cause colour changes as the object moves.

FILL background	*FILL foreground*	*Colour*
16	∅	Black
17	1	Red
18	2	Green
19	3	Yellow
20	4	Blue
21	5	Magenta
22	6	Cyan
23	7	White

Fig. 9.13. Colour codes for the FILL instruction.

they include a sound system that is unmatched at its price, and not exactly overshadowed by machines at several times its price. One of the most useful features for the writer of games – and not to be sneezed at by the business user either – is the set of pre-programmed sounds. That means what it says – you can get a sound just by typing a name. The four pre-programmed sounds are illustrated in Fig. 9.14. You can use any of them as a direct command (type ZAP and press RETURN) or within programs, as:

```
    1ØØ EXPLODE
```

or

```
    12Ø IF A = Ø THEN SHOOT
```

Command	*Effect*
PING	Bell sound to attract attention.
ZAP	Useful for laser rays, death rays, phasor guns ...
SHOOT	Single revolver shot.
EXPLODE	Reverberating noise like prolonged SHOOT.

Fig. 9.14. The pre-programmed sounds of ORIC.

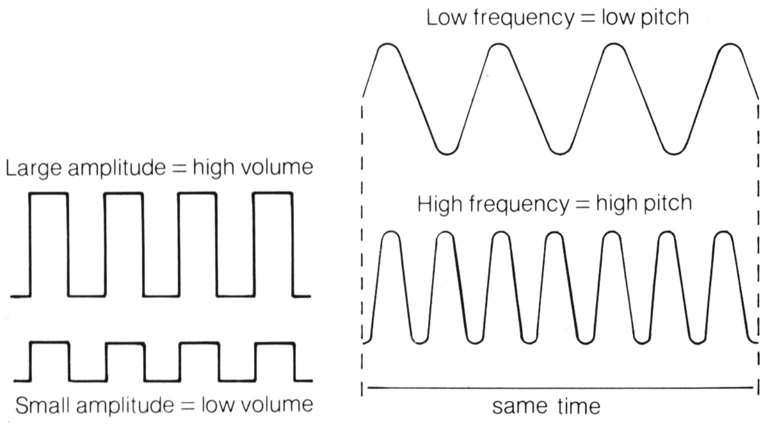

Low frequency = low pitch
Large amplitude = high volume
High frequency = high pitch
Small amplitude = low volume
same time

Fig. 9.15. The amplitude and frequency of a sound.

Because these are so very easy to use, it's a great temptation to use only these effects, and to ignore the really excellent capabilities of the ORIC for generating sound by programming of your own. The three instruction words that we need are PLAY, SOUND and MUSIC but, before we look at them, we need to understand something about sound itself and how we produce what we call music.

To start with, the sound that reaches our ears is caused by vibration of the air. This vibration of the air is in turn caused by the vibration of something in contact with the air, like a string, a drumskin, the lips of a trumpeter, and so on. The loudness of the sound depends on the amount of vibration – its *amplitude* as we call it. The pitch of the sound – low or high – depends on the *frequency*, which is the number of vibrations per second (see Fig. 9.15). The vibration also has what we call a *waveshape* or *waveform*. If we could plot a graph of how the vibrating material moved over the time of a few cycles of vibration, we would see shapes like these in Fig. 9.16. Of these waveshapes, the sine wave is the simplest. It is produced when you whistle, but it's a sound that your ears tire of

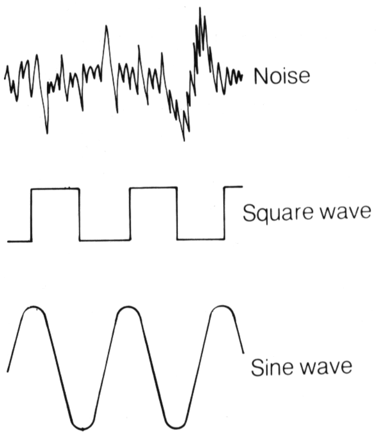


Fig. 9.16. The waveshape of a sound wave.

very quickly. Electronic instruments produce a waveshape more like a square wave, which gives a more interesting sound. We can also have noise, which has no regularity about its waveform.

The waveform is only part of what makes a musical note. A musical note contains a large number of waves, and we usually find that the amplitude (loudness) is not constant all the way through a note. The standard method of analysing a musical note is into an *envelope* with attack, decay, sustain and release portions (Fig. 9.17). The idea is that a musical note starts with a build-up of volume, it dies away to some extent, remains steady for a short time, and then cuts off. All of this, however, is difficult to program.

ORIC gets around this by offering you a set of ready-made 'envelopes' and we shall look at these later. For the moment we need to consider one more point about music – the *musical scale*. The traditional scale for Western music uses twelve notes, called a

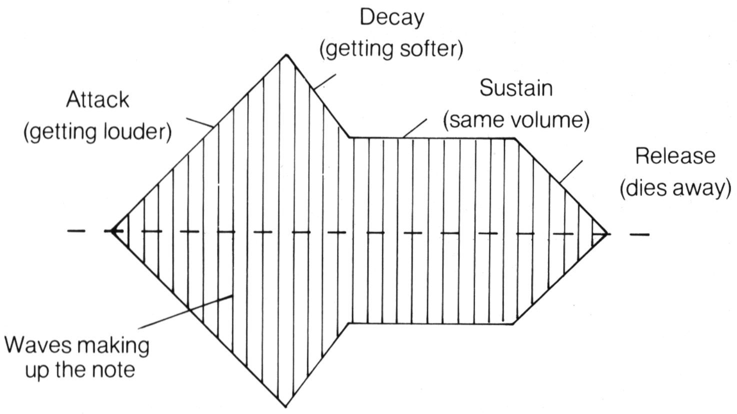


Fig. 9.17. The 'envelope' of a music note, containing many waves.

chromatic scale, but for many purposes we write music using a set of eight notes – the *octave*. The notes are lettered, and their pitch is indicated on a diagram such as Fig. 9.18 showing the staves of music. A *stave* means a set of five lines and four spaces, and the most important note for our purposes is the one located between the treble (upper) and bass (lower) staves – *middle C*. The name is appropriate because this is the note between the staves, and also because it is the note at the middle of a piano keyboard (Fig. 9.19). It's also the note which is around the centre of the ORIC's range of musical notes.

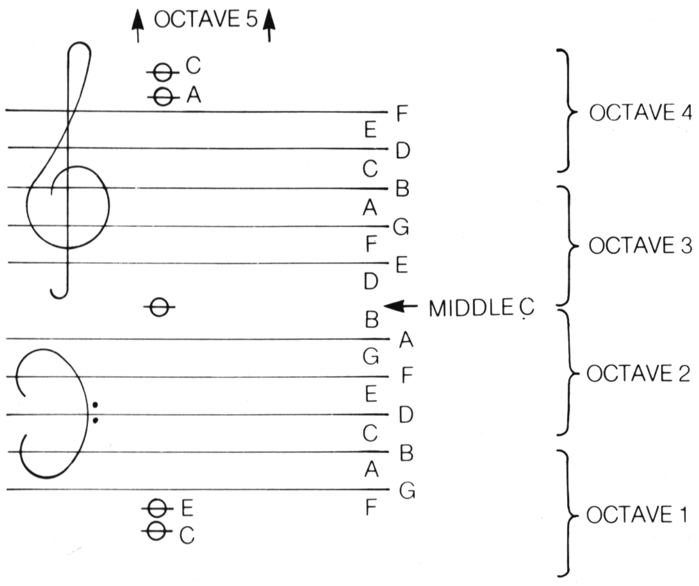


Fig. 9.18. How the pitch of a note is indicated in music.

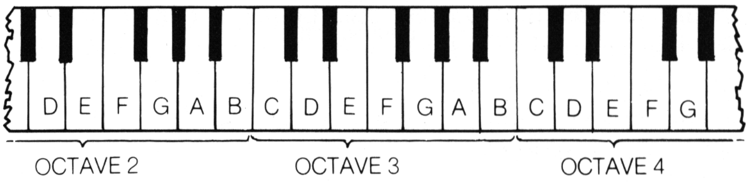


Fig. 9.19. How the piano keyboard relates to ORIC notes.

SOUND alone

How do we program the ORIC for sound then? To start with, it depends on what we want to do. One instruction is used simply to produce notes that we can use as warning notes. It is the SOUND instruction, and it has to be followed by three numbers. The first number is a 'channel' number. The ORIC allows you to program up to three notes together (plus noise), so there are three note 'channels' that can be controlled separately. The SOUND instruction will use only one channel at a time, but we can mix noise in with the note to produce rather interesting effects. The second number is a period number, and a value of 238 here will produce a note which is about middle C. Doubling the period number produces a note which is one octave below middle C; halving the value (to 119) gives an octave above middle C. It takes some time, if you have been thinking in terms of frequency to become used to the idea that the higher number gives lower notes.

The last number in the SOUND instruction is the volume number, allowing you to control the volume of sound to the ORIC's internal (and very effective) loudspeaker. This is one of the few loudspeakers on computers which really does deserve its name!

As an introduction, we can use SOUND as a direct command – the note will continue to play until you press another key. Try:

```
SOUND1,238,2
```

which gives middle C at a fairly low volume. Only Channel 1 can be used if you have not previously permitted the other channels to be 'opened' by the PLAY instruction, but we'll come to that later. Meanwhile, try the program in Fig. 9.2∅ which alters the volume of the note as it plays. Then try Fig. 9.21, which alters the pitch on the way through. Because of the way that the period numbers affect the pitch, you'll find that the changes are very small at first, but very much more noticeable as the program nears an end. A better effect

```
10 FORN=1TO15:SOUND1,238,N
20 FORJ=1TO50:NEXT
30 NEXT
40 FORJ=1TO50:NEXT
50 FORN=15TO1STEP-1:SOUND1,238,N
60 FORJ=1TO50:NEXT
70 NEXT
```

Fig. 9.20. A program that varies the volume of a note as it sounds.

```
5 REM PITCH
10 FORN=4096TO1STEP-1
20 SOUND1,N,2
30 NEXT
```

Fig. 9.21. Altering the pitch of a note as it sounds.

can be obtained if you use a different starting value of N – 16∅∅ is better than 4∅96.

Now we get to more complicated sounds. The SOUND instruction by itself employs Channel 1 of the ORIC sound system, and if we want to make use of other channels, and of noise mixed with the note, then we have to use PLAY. PLAY is an 'enabling' instruction which allows you much more control over the other instructions. Let's see how it works.

Play it again, Sam

PLAY is followed by four numbers. The first is a *tone enable number*, range ∅ to 7, whose actions are listed in Fig. 9.22. We use it to connect up the channels that we want to use, in particular when we want to use any channel other than 1, or when we want to mix

Tone enable	*Effect*
∅	No tone channels on – other sound instructions have no effect.
1	Channel 1 on.
2	Channel 2 on.
3	Channel 1 and Channel 2 on.
4	Channel 3 on.
5	Channel 3 and Channel 1 on.
6	Channel 3 and Channel 2 on.
7	Channels 1, 2 and 3 all on.

The same codes are used for the noise channels.

Fig. 9.22. The tone enable codes.

channels. The second number of the PLAY instruction is a *noise enable*, which uses the same numbers as the tone enable. These numbers allow noise to be injected into any channel or mixture of channels. The third number, the *envelope mode*, lets you select from any of the envelope shapes that are illustrated in Fig. 9.23. The fourth number, the *envelope period*, controls the duration of the sound in the envelope. Now we need to look at these examples.

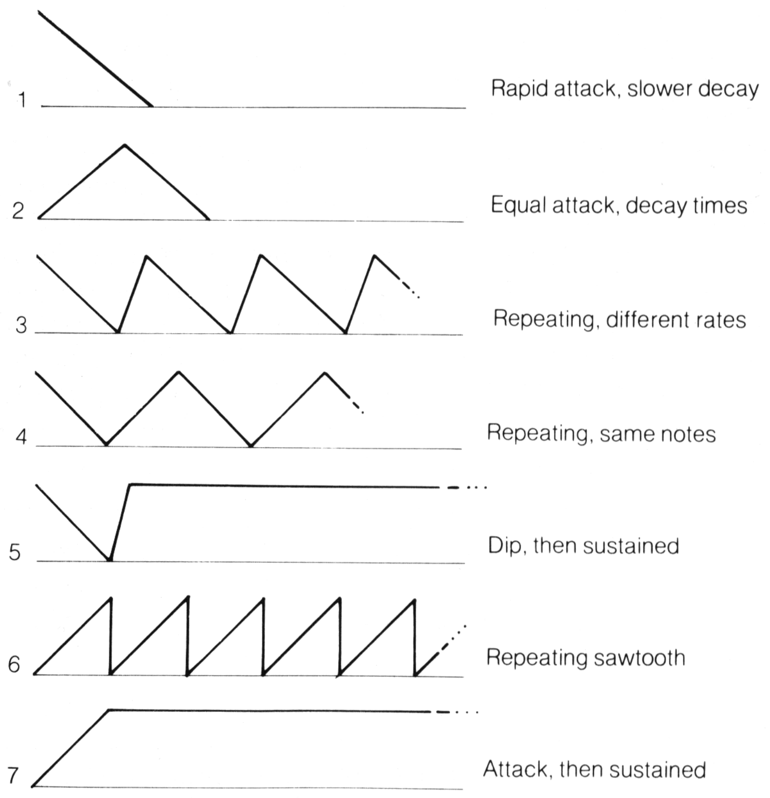


Fig. 9.23. The ORIC selection of envelope shapes.

Fig. 9.24 uses PLAY to open channel 1 to sound and also to noise, with envelope type and length set to ∅. The SOUND instruction in line 2∅ uses 4, which means tone channel 1 mixed with noise. If we wanted tone channel 2 or 3 mixed with noise, then we need SOUND 5 or SOUND 6, but only if the channels have been allocated by

```
5 REM PLAY ENABLE
10 PLAY1,1,0,0
20 SOUND4,238,5
```

Fig. 9.24. Combining a note with noise.

PLAY. The program has opened both sound and noise to channel 1, so that we should hear both simultaneously. We can, if we like, reverse the order of lines 1∅ and 2∅ so that the SOUND instruction sets up the note and the PLAY instruction enables the SOUND to be heard.

Now how about some harmony? At this stage, SOUND becomes less attractive to use, because the number that is used for the period in the SOUND instruction is not so easy to relate to the pitch as the numbers in the MUSIC instruction. We use SOUND when we want to make sounds that will zap or whistle over a large range of frequencies. 'Horses for courses' is the motto, and MUSIC is the instruction for melody and harmony.

MUSIC uses four numbers. The first one is the *channel number*, which can be 1, 2, or 3 for tones alone, or 4, 5, or 6 when the tones are mixed with noise. The second number is the *octave number*, ranging from ∅ to 6, low number meaning low notes. On this scale, octave 3 starts with middle C. This is an unusually large range of notes for a computer in this price range, and yet another reason for preferring ORIC! The third number is the *note number*, which follows the scheme listed in Fig. 9.25 – one number for each note in the

Number	*Note*	*Number*	*Note*
1	C	7	F#
2	C#	8	G
3	D	9	G#
4	D#	10	A
5	E	11	A#
6	F	12	B

Fig. 9.25. The note numbers for the MUSIC instruction.

chromatic scale. Note that these numbers are the same in each octave, so that the MUSIC instruction is not so useful if you want, for example, a wailing note covering more than one octave. The fourth number is for *volume* on the usual ∅ to 15 scale.

As before, the MUSIC instruction can be placed before or after the PLAY instruction. In other words, you can have a MUSIC instruction and then PLAY the music, or you can have a PLAY instruction followed by the MUSIC.

Take a look at Fig. 9.26. The PLAY instruction in line 1∅ opens channels 1 and 2, but does nothing else. The MUSIC instruction in line 2∅ then plays a note – middle C at the start of octave 3, and the

```
5 REM HARMONY
10 PLAY3,0,0,0
20 MUSIC1,3,1,5
30 MUSIC2,3,6,5
```

Fig. 9.26. Using two channels to produce harmony.

next MUSIC instruction in line 3∅ plays F in octave 3, giving a harmony. So far, so good. The next thing we might want is some rhythm. Rhythm is produced by a brief burst of noise, using the SOUND instructions. The period number is now limited to the range of ∅ to 31, and it will decide what the noise sounds like – a bass drum, side drum, snare drum or whatever. Fig. 9.27 shows an

```
5 REM RYTHM
10 FORN=1TO8
20 READJ
30 PLAY1,0,1,50
40 MUSIC1,3,J,5
45 WAIT10
50 PLAY0,1,1,40
60 SOUND1,10,5
65 WAIT10
70 NEXT
80 PLAY0,0,0,0:REM STOPS SOUND
90 END
100 DATA1,5,8,10,11,10,8,5,1
```

Fig. 9.27. Rhythm added into a music program by using bursts of noise.

illustration of this in action. The MUSIC instruction decides the notes, the SOUND produces the noise, and PLAY switches the channels between tone and noise so that you don't have both continually. All of the notes are in one octave, which considerably simplifies the programming.

Now for the finale – a piece of three-part harmony! We select all three channels, and play different notes on each. The program is in Fig. 9.28. This time the MUSIC is set to give the notes, and the volume is set by the envelope mode that we use in the PLAY instruction. To do this, we must set the volume in the MUSIC instruction to zero, allowing the volume then to be controlled by the PLAY envelope shape. The notes that are played, and the duration of each WAIT, are decided by the DATA that is stored in the DATA lines. I have used each DATA line to hold the numbers for four notes, which makes it relatively easy to sort out where a problem occurs. If you are working from published music, use each DATA line to hold the numbers for one complete bar of music. A *bar* is the amount of notes that will fit between the vertical bar divisions on the staves.

```
 5 REM NEW YEAR
 10 FORN=1TO12:GOSUB1000:NEXT
 20 FORN=1TO2:GOSUB2000:NEXT
 100 END
 1000 READT1,T2,T3,P
 1010 MUSIC1,3,T1,0
 1020 MUSIC2,2,T2,0
 1030 MUSIC3,1,T3,0
 1040 PLAY7,0,2,700
 1050 WAITP
 1060 RETURN
 2000 READT1,T2,T3,P
 2010 MUSIC1,4,T1,0
 2020 MUSIC2,3,T2,0
 2030 MUSIC3,1,T3,0
 2040 PLAY7,0,2,1000
 2050 WAITP
 2060 RETURN
 3000 DATA1,1,1,60,6,6,6,90,6,6,4,30,6,1
2,6,60
 3010 DATA10,12,6,60,8,1,1,90,6,1,1,30,8
,1,1,60
 3020 DATA10,12,1,60,6,6,6,90,6,10,1,30,
10,1,1,60
 3030 DATA1,1,6,60,3,3,12,120
```

Fig. 9.28. The ORIC in three-part harmony!

This program isn't an end – it's just a beginning. Though the notes are taken from a sheet music edition, they don't sound quite right. My early ORIC has a habit of introducing unwanted and unprogrammed high notes on Channel 3 (good thing it doesn't have a Channel 4), and this will undoubtedly have been sorted out on later machines. In addition, the small loudspeaker cannot really cope with the low pitch (bass) notes in this program. Playing through a hi-fi system will produce much better effects.

There it is, then. Welcome to the world of ORIC – it's one that should keep you exploring new wonders for many years to come. All we have to do now is to tidy up some loose ends in Chapter 10 and the Appendices, and then you're on your own, a beginner no longer.

Chapter Ten

Odds and Ends

No matter how thoroughly we look at a set of computer instructions, there are always a few that don't seem to fit under any heading that we use. We'll start this chapter by looking at a few ORIC instructions that come under this heading.

One such instruction is CALL. CALL is used to 'call up' a special type of subroutine called a *machine code* subroutine. Machine code is a number code that is used to control the 6502 microprocessor that is the 'brain' of the ORIC. When we use BASIC, each instruction word is used to locate and call up a large amount of this machine code, but the use of CALL allows us to use parts of the code that are not used by BASIC, including parts which we place into memory from tape. The snag is that unless you know and understand machine code, and know what the addresses in memory are used for, you can't make much use of this instruction. Since the use of machine code would need at least another book of this size, there is little point in devoting much space to the use of CALL. The WELCOME TO ORIC tape that came with my ORIC illustrated the use of CALL in the form of CALL@FB∅3 which produces a key click in the middle of a program at a time when you haven't touched a key.

CLEAR is an instruction which clears the memory of all variable values. Number variables are set to zero, and string variables to a blank or empty string. We seldom need to use CLEAR in a program, and the CLEAR action is carried out when we type RUN and press RETURN in any case.

FRE is a rather more useful instruction. When FRE is followed by ∅ in brackets (as in PRINT FRE(∅)), it gives the amount of memory, in bytes, that remains unused. It's a useful way of finding how much memory you have left after you have entered a long program, for example, and from the value you get, you can work out how much memory your program has used. Another use is in a data processing program where there is a chance, particularly if you are using the 16K machine, that you might run out of memory. This can be avoided if, before entering each item, the amount of free memory is measured, so that a warning can be given if memory is running out. You might, for example, use a line such as:

```
IF FRE(ø)<2øø THEN PRINT "OUT OF SPACE – PLEASE
MAKE THIS THE LAST ITEM"
```

Another use for FRE is in 'garbage collection'. During a long data processing program, the same variable name may be used for a large number of different values. This is particularly true when an alphabetical sort is used. The computer uses new memory space every time a variable is reassigned, and it does not clear the unused values, normally, until the memory is full. When this happens, all computing stops, and the variables are shuffled around, clearing out all the unwanted values. This can cause some of your programs to seem to hang up, with the computer producing no screen messages, not responding to keys, and generally behaving as if it had just gone fifteen rounds with King Kong. A program which gives problems of this type can be greatly improved by adding a FRE("") instruction at intervals. Using FRE("") frees the unused variable names, and carries out a minor re-organisation of memory. If this is done before the memory is full, it takes very much less time (there is very much less to sort out), and it greatly speeds up the action of the computer on this type of program.

GRAB is another instruction that is useful if you are running long data processing programs, particularly on the 16K machine. GRAB allows your program to use the part of the memory that is usually allocated to the screen when the HIRES instruction is used. If your program does not use high resolution graphics, then GRAB can get you quite a lot more useful memory. Some machines do not permit this action – you are stuck with the smaller amount of memory whether you use high resolution graphics or not. There is an opposite instruction, RELEASE. This has to be carried out if you have used GRAB and you now want to make use of the high resolution screen. Be careful of this, though. If you happen to have something, which might be part of your program, stored in this part of the memory at the time when you use RELEASE, you will lose it.

STR$ is a string function which converts a number into a string. For example, if A = 27.6, then by using B$ = STR$(A), we assign B$ just as if we had typed B$ = "27.6". This can be useful at times if we want to use instructions like LEFT$, RIGHT$ and MID$ on numbers. It's easy, for example, to line up numbers into columns when the numbers are in string form, because we can use LEN to find the number of characters, and then make use of this number in TAB instructions. A point to watch, though, is that STR$ always puts a blank space in front of the number value. This is to leave room for a + or − sign, but it can cause you some confusion if you are using LEN, and you find that the number 1 converted to string form now has two characters!

Expanding ORIC

For a lot of purposes, the combination of ORIC-1, a TV receiver and a cassette recorder is perfectly adequate. Like all serious computers, however, ORIC can be used with a much greater range of equipment. If you start to get really serious about your computing, you will want to know how ORIC can be expanded into a big and even more capable computer system.

Memory is usually the first part of expansion, if you have started with a smaller-memory machine. The memory of the computer, remember, is measured in bytes, and 1024 bytes make up the larger unit, the kilobyte, or K. One byte is the amount of memory that we need for storing one single typed character. The standard memory sizes for the original ORIC were 16K and 48K, though a 32K machine was announced about a month later. The 48K machine which I have used in the preparation of this book has a massive amount of available memory – more than that of some machines which are advertised as having 56K or even 64K! If your ORIC is the 16K model, however, you may eventually find that you want to increase your memory size, but only if you want to use very long or elaborate programs.

Another upgrade item is a video monitor. A TV receiver is adequate for a lot of purposes, but the letters and drawings on the screen are never as clear and well-focussed as they could be. This is because of the limitations of the TV receiver, not because of the ORIC. By using a video monitor, which takes the signals direct from the ORIC rather than in the way they have to be coded for TV transmission, pictures with greatly improved focus and colour can be obtained. Both colour and B/W monitors can be bought, though the colour monitors are very much more expensive. This is because the colour monitor is much more difficult to manufacture to the acceptable standard of picture quality. Several manufacturers are at present bringing out TV systems which use a separate monitor. This allows you to update other parts of the system (like a hi-fi) without replacing others, and such monitors *may* be suitable for use with ORIC. Be careful, though; not all types of monitors can be connected. A user group can be useful in this respect by advising ORIC owners which monitors are suitable.

The third item of upgrading is a disc system. Cassettes are useful for storage of programs or data, but the cassette recorder has to be started and stopped by you, and you have to find the correct place on the cassette. Cassettes are slow, too. Each side of a C-60 cassette needs 30 minutes to record or replay, and computers are intended to save time, not waste it. A disc system uses a thin plastic disc which is held in a cardboard sleeve. The disc is coated with magnetic material, just like cassette tape, but the resemblance ends there. When the disc is placed in a disc-drive, signals from any part of the disc can be replayed, and the complete mechanism is controlled by the computer. Once you have switched on the disc drive and put in the disc, the computer takes over command. When you command a program to be loaded, the disc spins, the 'reading head' moves over the disc to find the program that you have requested and loads it. The head then retracts, and the disc stops. The whole process is complete in a few seconds.

Don't feel that you have to go out and buy a disc system right away, because that's running before you can walk. You should be aware, though, that all of these goodies will be available for the ORIC-1. This makes the ORIC an ideal choice, because as your computing needs grow, so also can your ORIC system.

User groups

When you are serious about your computing, membership of a user group is an essential. A user group consists of uncannily friendly and hard-working people who will spare any amount of time and effort to find out as much as they can about the computer of their choice. By attending user group meetings and reading user group magazines, you will find out as much about ORIC as anyone can ever know, and get very much more from your ORIC in consequence. At the time of writing, no specific ORIC user group had been set up, but it's possible that the Tangerine User Group (T.U.G.) would be interested in forming an ORIC section. Look for the address in the magazines.

Magazines

Magazines are the other way of keeping in touch with what's going on in the world of computing generally, and ORIC in particular. It's in the magazines that you will see the goodies advertised, telephone adaptors and other hardware, cassettes, discs and other software. It's also in the magazines that you will find lots of hints and tips. Remember that the BASIC of the ORIC is of the Microsoft type, so that a lot of programs written for the computers that use Microsoft BASIC will also be usable, with some modifications for graphics and sound, for the ORIC.

Of all the magazines, *Personal Computer World* is the granddaddy and carries the biggest range of useful advertisements. A fast-growing magazine which leans more to games and amusements is *Your Computer*, but if you are interested in the hardware side of the business as well as in programming then magazines like *Electronics and Computing* and *Computing Today* will be of interest.

Appendix A
Cassette Loading Problems

Cassette recorders, like open-reel recorders, work on the principle of pulling a tape (plastic-coated with iron oxide) past a tapehead, which is a miniature electromagnet. The important part of any tapehead is the 'gap', a tiny slit in the metal, too fine to see except under a microscope. This slit should be tilted so that it is at 90° to the edge of the tape, but this angle is often quite a long way from 90° even when the recorder is new. A poorly set up tapehead will make it difficult to load programs that have been recorded on correctly set up equipment, though you will usually be able to load tapes that you have recorded using the same machine. *Never* touch the tapehead with anything made of metal. You can, however, set up the head to the correct angle using the scheme outlined here. Once you have set up the head correctly, there should be no need to change the setting again unless an odd tape refuses to load. Check before you re-set the head, however, that the trouble is not caused by dirt. Buy a tapehead cleaning kit and use it as per the instructions.

Head alignment

You have to start by looking for the head-adjusting screw. The Trophy CR100, for example, has a small hole drilled in the top of the case, just under the cassette lid (Fig. A1). If you insert a jeweller's screwdriver through this hole when a tape is playing, the screwdriver can be engaged in the adjusting screw. First of all, find a cassette which refuses to load. Pull out the small motor control jack plug to give you control over the cassette motor, and allow the sound to be heard. Put the cassette in and play it. Listen to the note. A cassette that refuses to load will usually give a muffled sound, and you have to adjust the head-adjusting screw until this sound is sharp. You may have to turn the screw either clockwise or anti-clockwise. If

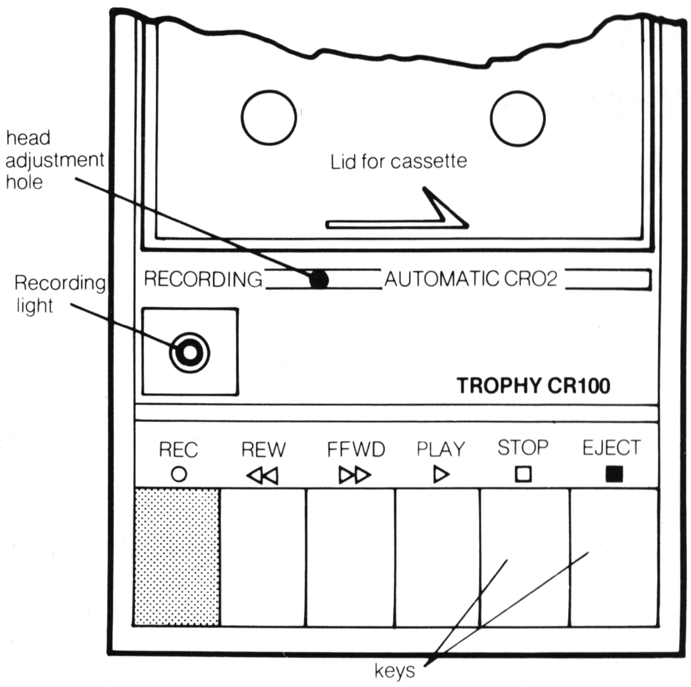


Fig. A1. The adjustment hole for the head of the CR100 cassette recorder.

a half turn in one direction does not sharpen the sound, or makes it more muffled, then turn in the other direction. You should be able to reach a spot where the sound is quite definitely sharper, more brilliant and with lots of treble. Shut off, restore normal operation, and try to load the cassette. You should now find that it loads. I have never found a cassette that did not respond to this treatment, unless it had been partly erased by being near a magnet. One other failure was due to a notorious computer which did not record its own cassettes correctly!

Not all recorders have the adjusting screw so well placed as that of the CR100, and you may have to remove the cassette lid to reach the screw. In one case (a very cheap recorder) I had to drill a hole for myself, but this is unusual. The higher the quality of the recorder, the more likely that the adjustment will be easy to reach!

Appendix B
Editing

Editing means altering a line that has been previously entered into the memory of the computer, and it introduces several special commands that were not available on my very early ORIC. However, the principle is to place the cursor on the line that is to be corrected, to move it to the point where changes are needed, and then to use commands that will insert, change or delete a character.

Editing is started by finding the line which contains the mistake. This may be immediately after entering the line, or later when you LIST or possibly when you RUN the program and find an error message. You have to start editing by typing:

    LIST line number

For example, if your error was in line 5ø, you would type LIST5ø. When you press RETURN, the ORIC prints the line, 5ø in this example, and you can now make a limited number of changes. If the line is a short one, it's really always quicker and easier just to retype the line, because a new line 5ø will replace the old one whenever you press RETURN at the end of it. If, however, you have a longer line which needs the correction or deletion of a character, then the rather elementary editing procedure of the early production ORIC can be used. Use the up-arrow key to place the cursor at the same level as the line number. Now place the cursor over the mistake by pressing CTRL A. Do *not* use the right-arrow key, because this has no editing effect.

When the cursor is over the place that you want you can replace a character by typing a new character. If you want to delete a character, you'll have to ensure that the cursor is placed at the immediate right-hand side of the character. Pressing the DEL key will then back-space the cursor and delete the character. If you want to return to normal, you then have to use CTRL A to place the cursor at the end of the line and then press RETURN. If you don't put the cursor to the end of the line, anything under and to the right of the cursor will be deleted when you press RETURN. This is a useful way of hacking off the unwanted remainder of a line. You may, for example, have a multistatement line from which you want to remove the last statement.

At the time of the production launch of ORIC there was no indication when more comprehensive editing facilities would be available or what syntax might be used.

Appendix C
Using a Printer

The ORIC-1 provides a 'parallel Centronics' connector for a printer. This consists of a rectangular socket which requires a suitable cable for the printer you intend to use. The listings in this book were prepared with my Epson MX-80, using a printer cable which was originally made for a Dragon computer! There should be no problem, therefore in connecting the ORIC to any normal parallel printer.

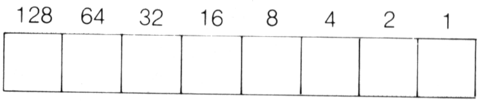
A few points need noting, however. One is that the ORIC is set internally to limit printed lines to 8Ø characters, to match the screen lines. This can cause a PRINTER ERROR message to appear at times and this can be avoided if the command:

```
POKE 49,255
```

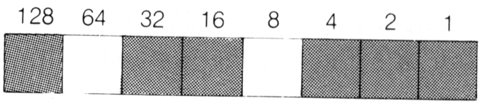
is carried out before printing. It is also an advantage to set the printer to operate with 40 characters per line. The POKE instruction allows up to 255 characters to be sent to the printer in a line, though it has no effect on the limitation of 8Ø characters in a screen line of BASIC (two lines on the screen).

For listing a program, the command word that you need to use is LLIST, which will produce printed copy of the whole program. You can also use all the variations that can be used with LIST, such as LLIST1ØØ, which will print one line, LLIST1ØØ – 2ØØ, which will print lines 1ØØ to 2ØØ and so on. To obtain a print-out of data from a program, use LPRINT in place of (or as well as) PRINT.

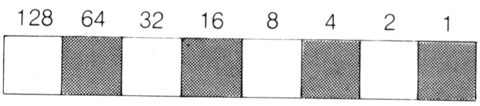
Appendix D
Creating Your Own Patterns



The PATTERN instruction will cause a pattern of dots and dashes to appear in a line drawn by DRAW or CIRCLE. The number that is used to follow PATTERN is determined by the shape above. Each section of line is divided into eight parts, as shown. You decide that your PATTERN shape will be by filling in some of these parts, thus:



You then add up the numbers above the shaded boxes; this number is then the PATTERN number. For example:



64 + 16 + 4 + 1 = 85 so that
PATTERN 85 would produce this
(dotted) line

Index

CONSULT THE ORIC!

The ORIC-1 is causing a sensation amongst microcomputer enthusiasts. It has a large memory, uses a familiar and well-established version of BASIC and can utilise a variety of add-on facilities. It provides superb colour, high resolution graphics and sound. ORIC graphics are Prestel/Viewdata compatible, and the ORIC can also be used with a modem to give a host of communications facilities.

This book for aspiring adepts introduces Microsoft BASIC, thoroughly explains the ORIC's graphics, colour and sound systems and also sets out the data processing capabilities to open up the full range of the ORIC's facilities to the beginner.

Many examples of useful programs are included. You will continue to use this book as a convenient reference long after you have mastered the ORIC's latent power!

The Author
Ian Sinclair is a well-known and regular contributor to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics,* and *Electronics Today International*. He has written some forty books on aspects of electronics and computing, mainly aimed at the beginner.

Front cover photograph by permission of
ORIC Products International Ltd.

GRANADA PUBLISHING
Printed in Great Britain 0 246 12130 0

£5.95 net

SINCLAIR **THE ORIC-1**

GRANADA